ICS 222: Basic concepts of computation UHM

3: State, coinduction

By: Dusko Pavlovic Honolulu

Reminder. You studied state machines in the prerequisite courses. This is a refresher lecture. This conceptual background will be useful in the present course, but is not central for it.

The sections printed in smaller fonts provide background information, and indicate extensions for interested students.

Contents

1 Algorithms for words

To represent the number 17, Lucy needs to count to 17, by fingers and by toes. To add 83 by recursion, she must count all the way up to 100. She could do that by fingers and toes as well, provided that she uses her fingers to count to 10, and her toes to count how many times she counted to 10. But once she knows how to represent numbers in this way, she could also compute 17+83 in just 2 steps, instead of 100. She would use the 2 steps to process the numbers 17 and 83 as 2-letter *words* in the language of decimal *numerals*. This is the language we all learn to speak at school. Speaking it means applying the word-processing algorithms. Speaking it fluently means not paying much attention to the algorithms. But computers need to know their algorithms.

1.1 Numbers as words: The numeral systems

Counting with her 10 fingers, Lucy can count up to 10 apples. If she uses the 5 fingers of the right hand to count the apples, and the 5 fingers of the left hand to count how many sets of 5 apples she counted, then she can count up to $5^2 = 25$ apples. If she uses each finger as a digit[1] of a binary numeral, then she can count up to $2^{10} = 1024$ apples.

For any integer $\beta \geq 2$, a *numeral in base* β is a sequence $\boldsymbol{a} = (\, a_m\, a_{m-1} \cdots a_2\, a_1\, a_0\,)_\beta$ of integers $0 \leq a_m, \ldots, a_0 < \beta$. It represents the number

$$\overline{\boldsymbol{a}} \quad = \quad a_m\beta^m + a_{m-1}\beta^{m-1} + \cdots + a_2\beta^2 + a_1\beta + a_0\beta^0 \tag{1}$$

Archimedes showed that every number can be represented in this way. When the base β is clear from the context, it is omitted, and the numeral is written as the sequence $a_m a_{m-1} \cdots a_2 a_1 a_0$. When Lucy counted 17 apples, we write 17_{10} without the subscript 10 because the base 10 is assumed as the default. In computers, the default is the base 2, and computer would thus write 10001 for $1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 17_{10}$. The binary numerals $(\, a_9\, a_8 \cdots a_2\, a_1 a_0\,)_2$ capture the numbers between 0 and 1023. If Lucy starts counting from 1, and not from 0, then she can count 1024 things using her fingers as 10-digit numerals in base 2. For $\beta = 5$, the numerals $(\, a_1\, a_0\,)_5$ can represent the numbers a

[1] Remember that *digitus* means finger in Latin.

between 0 and 24. If Lucy counts the apples by her right hand, and uses her left hand to count the right-hand counts, then the right hand is a_0 and the left hand is a_1. The fingers correspond to the digits $0, 1, 2, 3, 4$.

The numerals in base β are thus the words in the alphabet $\mathbb{Z}_\beta = \{0, 1, \ldots, \beta - 1\}$. The arithmetic operations on the set of all integers $\mathbb{Z}$ are implemented by *word processing* over the alphabets $\mathbb{Z}_\beta$. The modular arithmetic modulo provides the operations on the digits. The operations on the words are derived for the representations (**??**). This goes through even for the transfinite numbers, which still have the normal forms (1), albeit for infinite bases.

Counting can be viewed as the implementation of aritmetic in $\beta = 1$. The modular arithmetic over $\mathbb{Z}_1$ is, of course, trivial, as its only element is 0. But if its only element is taken to be 1, then (1) boils down to (**??**), since

$$n \;=\; 1 \cdot 1^{n-1} + 1 \cdot 1^{n-2} + \cdots + 1 \cdot 1^0 \;=\; \underbrace{1 + 1 + \cdots + 1}_{n \text{ times}}$$

The number n is thus captured by length of the numeral $11 \cdots 1_1$. Using their fingers to represent the digit 1, children count to 10 in this way. If the fingers are used as the *binary* digits, then each finger represents a fixed *position* of a 10-digit numeral, and each position can be filled by the digit 0 or 1. This is captured by the two *states* of the finger, usually *bent* for 0 and *straight* for 1. In all cases, the fingers are used as the *external memory*: they record a digit as their state. The difference is that, in the binary system, each finger has its own state, 0 or 1, whereas in the unary system, all fingers together record one of the states between 1 and 10. When a 2-digit numeral in base 5 is recorded using the fingers of two hands, then each hand carries its own state, from 0 up to 4, and this state is recorded by the fingers of that hand together.

1.2 Lists and words

A *list* is a finite tuple, possibly empty. A *word* is a nonempty list. For any set A, the sets of words and of lists are respectively

$$A^+ = \coprod_{n=1} A^n \qquad\qquad A^* = \coprod_{n=0} A^n = \{()\} \cup A^+$$

where $\coprod$ denotes the disjoint union. The set A from which lists and words are formed is often called *alphabet*, and its elements are then called *letters*. The identifiers for lists and words are often in the form $\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{z}$. Theory of computation is formalized as a theory of languages.

Definition 1. *An* language $\mathcal{L}$ *over an alphabet* A *is a set of lists or words* $\mathcal{L} \subseteq A^*$.

Viewed as types, A^* and A^+ are generated by the type constructors

$$I \xrightarrow{()} A^* \xleftarrow{(::)} A^* \times A \qquad\qquad A \xrightarrow{(-)} A^+ \xleftarrow{(::)} A^+ \times A$$

This means that every word must be in the form $\boldsymbol{a} = \left(a_0 :: a_1 :: a_2 :: \cdots :: a_n \right)$. For the string representing the type A^+, this means that all data that in it are produced in the boxes (::) or (−), as displayed in Fig. 1, which input letters from A and output words. The data along the string A^* are similar: rooted in the empty list (), and

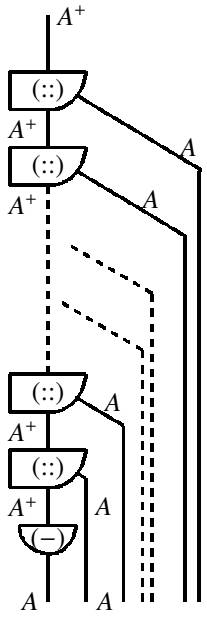
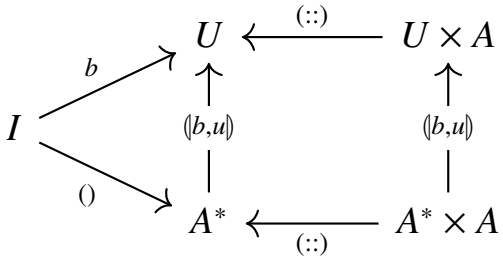


Figure 1: All words are constructed by the operations (−) and (::) as letter pumps

the letters are only appended through (::). Viewed as types A^+ and A^* are initial among the corresponding structures. For lists, this means that for all types U, any structure b, u induces a unique banana-function $(\!|b, u|\!)$ that makes the following diagram commute



For words, we spell this out in full detail, showing how it extends the induction over the natural numbers.

Channels are the functions in the form $c : A^+ \to B$. They are the *dynamic*, or

history-sensitive functions, in the sense that entering the inputs $a_0, a_1, a_2, a_3 \ldots$, into a channel and into an ordinary an ordinary, *history-free* function $f : A \to B$ leads to the following dependencies

$$\begin{array}{cc} a_0 \mapsto c(a_0) & a_0 \mapsto f(a_0) \\ a_0a_1 \mapsto c(a_0a_1) & a_1 \mapsto f(a_1) \\ a_0a_1a_2 \mapsto c(a_0a_1a_2) & a_2 \mapsto f(a_2) \\ a_0a_1a_2a_3 \mapsto c(a_0a_1a_2a_3) & a_3 \mapsto f(a_3) \\ \ldots & \ldots \end{array}$$

A channel, in a sense, has a memory, as its output depends on the past inputs, whereas an ordinary function does not have a memory, and its outputs at different moments in time are independent on one another. In theory of communication, channels are modeled as processes which depend on sources, which are histories as above, just random. We presently only consider deterministic channels.

An ordinary function $f : A \to B$ maps

$$a \;\mapsto\; f(a)$$

A channel consumes

1.3 Word induction

Definition 2. *The* word induction schema *over the* alphabet A *and the* word space W *is*

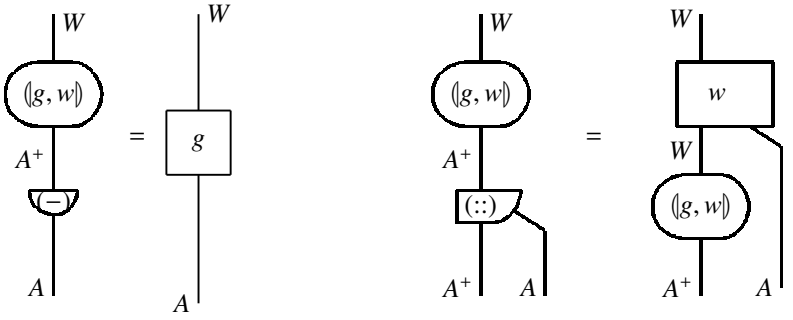
$$\frac{g\colon A \to W \qquad\qquad w\colon W \times A \to W}{(\!|g, w|\!)\colon A^+ \to W} \tag{2}$$

The channel $(\!|g, w|\!)$ *is defined*

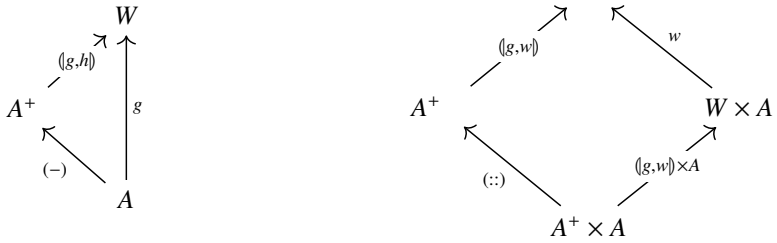
$$(\!|g, w|\!)(a) = g(a) \qquad\qquad (\!|g, w|\!)(z{::}a) = w\Big((\!|g, w|\!)z,\ a\Big) \tag{3}$$

The diagrammatic versions of (3) are aligned in Fig. 2.

$$(\!|g, w|\!)(a) \;=\; g(a) \qquad\qquad (\!|g, w|\!)(z{::}a) \;=\; w\Big((\!|g, w|\!)z,\; a\Big) \tag{3}$$



(4)



(5)

Figure 2: The three views of word induction

1.4 Running the word induction

A channel $(\!|g, w|\!)$ is evaluated letter by letter along (3).

$$\begin{aligned}(\!|g, w|\!)(\boldsymbol{x}) \;&=\; (\!|g, w|\!)\Big(\, x_0\, x_1\, x_2 \cdots x_{n-1}\, x_n \,\Big) \;= \\ &=\; w\Big((\!|g, w|\!)\Big(\, x_0\, x_1\, x_2 \cdots x_{n-1} \,\Big),\; x_n\Big) \;=\; w\Big(w\Big((\!|g, w|\!)\Big(\, x_0\, x_1\, x_2 \cdots x_{n-2} \,\Big),\; x_{n-1}\Big),\; x_n\Big) \;= \cdots \\ \cdots \;&=\; w\Big(w\Big(\cdots w((\!|g, w|\!)x_0, x_1) \cdots x_{n-1}\Big), x_n\Big) \;=\; w\Big(w\Big(\cdots w(g(x_0), x_1) \cdots x_{n-1}\Big), x_n\Big)\end{aligned}$$

The **string-diagrammatic** view of word processing is in Fig. 3. The **arrow-diagrammatic** view is in Fig. 4.

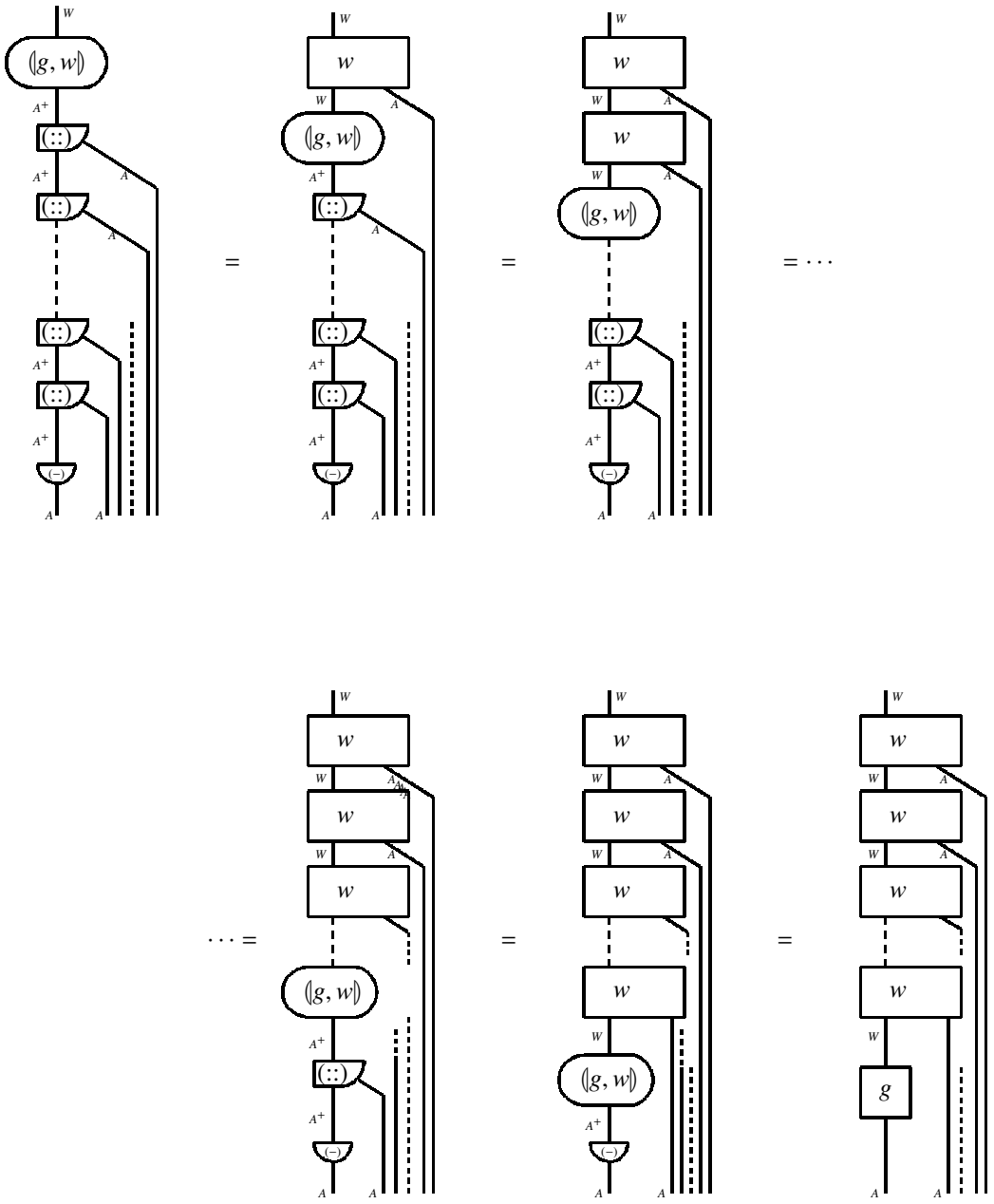


Figure 3: Evaluating an inductive channel

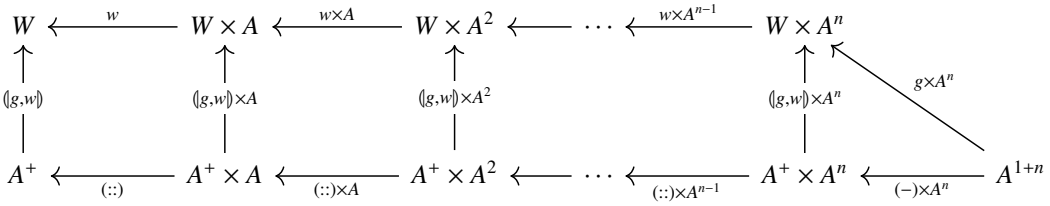


Figure 4: Evaluating a word-inductive function $(\!|g, w|\!)$ by chasing the arrows from left-up to up-left

1.5 Example: Decimal addition

Evaluating the recursive function $\alpha_n^{(1)}(m) = n + m$ from Sec. **??**, Ex. (a) on $n = 87$ and $m = 13$ to get $\alpha_{87}^{(1)}(13) = 100$ takes 88 steps of counting. Children's addition algorithm

$$\begin{array}{r} {}^{1}8\,7 \\ +\quad 1\,3 \\ \hline 1\,0\,0 \end{array} \tag{6}$$

takes two steps of word processing. This familiar procedure is an instance of word induction.

Declaring the types. The idea is to take

- the alphabet A to be the set of pairs of digits: e.g., in (6), the pair $\left(\begin{smallmatrix}7\\1\end{smallmatrix}\right)$ is processed as as the first letter from A, and then the pair $\left(\begin{smallmatrix}8\\3\end{smallmatrix}\right)$ is processed as the next one; and to take
- the word space W to be the set of triples of numerals $\left(\begin{smallmatrix}\boldsymbol{x}\\\boldsymbol{y}\\\boldsymbol{z}\end{smallmatrix}\right)$, all of the same length, such that $\overline{\boldsymbol{x}} + \overline{\boldsymbol{y}} = \overline{\boldsymbol{z}}$: in (6), we first form $\left(\begin{smallmatrix}07\\01\\10\end{smallmatrix}\right)$ in W, and then $\left(\begin{smallmatrix}087\\013\\100\end{smallmatrix}\right)$.

In summary, we thus take

$$\begin{aligned} A &= 10 \times 10 = 10^2 \\ W &= \coprod_{n=1}^{\infty} \left\{ \begin{pmatrix}\boldsymbol{x}\\\boldsymbol{y}\\\boldsymbol{z}\end{pmatrix} \in 10^n \times 10^n \times 10^n \;\middle|\; \overline{\boldsymbol{x}} + \overline{\boldsymbol{y}} = \overline{\boldsymbol{z}} \right\} \end{aligned}$$

where $10 = \{0, 1, 2, \ldots, 9\}$, $\coprod$ is the disjoint union, and $\overline{\boldsymbol{x}} = \sum_{i=0}^{n} 10^i x_i$ for $\boldsymbol{x} = (\, x_n\, x_{n-1}\, \cdots x_1\, x_0\,)$. The two steps of (6) should be

$$g\begin{pmatrix}7\\3\end{pmatrix} = \begin{pmatrix}07\\03\\10\end{pmatrix} \qquad\qquad w\left(\begin{pmatrix}8\\1\end{pmatrix}\,,\,\begin{pmatrix}07\\03\\10\end{pmatrix}\right) = \begin{pmatrix}087\\013\\100\end{pmatrix}$$

The first obstacle is a notational quibble: the induction step w is a function in the form $A \times W \to W$, whereas Def. 2 introduced it in the form $W \times A \to A$.

Left-to-right or right-to-left? Since this text is written left-to-right, the word induction has been presented in that direction, with the words in the form $\boldsymbol{a} = (\, a_0\, a_1 \cdots a_m\,)$, generated by the concatenation operation $A^+ \times A \xrightarrow{(::)} A^+$, which appends letters on the right. The step of the word induction was therefore also written as $W \times A \to A$. For better or for worse, the convention for the numerals is to write them starting from the most significant digits on the

left towards the least significant digits on the right. The addition and the multiplication algorithms, however, process numerals in the opposite direction, starting from the least significant digits on the right, and pushing the overflows to the left. The numerals are thus written in the form $\boldsymbol{a} = (\, a_m\, a_{m-1} \cdots a_0\,)$, with a_i corresponding to β^i. The digits processed later are prepended on the left by the concatenation operation $A \times A^+ \xrightarrow{(::)} A^+$. The corresponding inductive step is thus in the form $A \times W \to A$. This turns around the notation, but nothing essentially new happens in the process. The addition algorithm could be described in the exact notation of Def. 2, but the numerals would be written backwards. Our languages in general are collections of such quirks, and we tend to get used to them, rather than straighten them out. The addition algorithm is thus presented in the familiar form, as right-to-left word processing, in the notation *dual to Def. 2*.

Programming the functions g and w. The base case the word processing algorithm of decimal addition just adds pairs of single-digit numbers

$$
\begin{array}{rcll}
g\colon A & \longrightarrow & W & \\
\begin{pmatrix} n \\ m \end{pmatrix} & \mapsto & \begin{pmatrix} n \\ m \\ (n+m) \end{pmatrix} & \text{if } n+m < 10 \\
& \mapsto & \begin{pmatrix} 0{::} & n \\ 0{::} & m \\ 1{::}(n+m-10) \end{pmatrix} & \text{if } n+m \geq 10
\end{array}
$$

The step w should be something like

$$
\begin{array}{rcll}
w\colon A \times W & \longrightarrow & W & \\
\begin{pmatrix} n \\ m \end{pmatrix},\ \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \end{pmatrix} & \mapsto & \begin{pmatrix} n & {::}\boldsymbol{x} \\ m & {::}\boldsymbol{y} \\ (n+m) & {::}\boldsymbol{z} \end{pmatrix} & \text{if } n+m < 10 \text{ or} \\
& \mapsto & \begin{pmatrix} 0{::} & n & {::}\boldsymbol{x} \\ 0{::} & m & {::}\boldsymbol{y} \\ 1{::}(n+m-10) & & {::}\boldsymbol{z} \end{pmatrix} & \text{if } n+m \geq 10
\end{array}
$$

— *provided* that the sum $\overline{\boldsymbol{x}} + \overline{\boldsymbol{y}} = \overline{\boldsymbol{z}}$ does not incur a carry-over, and thus does not add 0s to $\boldsymbol{x}$ and $\boldsymbol{y}$ and 1 to $\boldsymbol{z}$ as the

last step. Taking this possibility into account, we get the complete definition of w as

$$w\left(\begin{pmatrix} n \\ m \end{pmatrix}\ ,\ \begin{pmatrix} n'::\boldsymbol{x} \\ m'::\boldsymbol{y} \\ r'::z \end{pmatrix}\right) \quad = \quad \begin{cases} \begin{pmatrix} n & ::\boldsymbol{x} \\ m & ::\boldsymbol{y} \\ (n+m+1)::z \end{pmatrix} & \text{if } \begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \ \wedge\ n+m+1 < 10 \\ \begin{pmatrix} 0:: & a & ::\boldsymbol{x} \\ 0:: & b & ::\boldsymbol{y} \\ 1::(n+m-9)::z \end{pmatrix} & \text{if } \begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \ \wedge\ n+m+1 \geq 10 \\ \begin{pmatrix} n & ::n'::\boldsymbol{x} \\ m & ::m'::\boldsymbol{y} \\ (n+m)::r'::z \end{pmatrix} & \text{if } \begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \ \wedge\ n+m < 10 \\ \begin{pmatrix} 0:: & a & ::n'::\boldsymbol{x} \\ 0:: & b & ::m'::\boldsymbol{y} \\ 1::(n+m-10)::r'::z \end{pmatrix} & \text{if } \begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \ \wedge\ n+m \geq 10 \end{cases} \tag{7}$$

Do children really compute like this? It looks complicated. The notation makes things more complicated than they are. The addition algorithm does perform the steps in (7), but children's implementation strips off the irrelevant details.

What information is relevant for adding numbers? The function w in (7) never tests the previously processed words $\boldsymbol{x}, \boldsymbol{y}$ and z. They are copied from the input to the output, but do not influence the rest of the output. Besides the input digits $\binom{n}{m} \in A$, the output of the function w only depends on the most recently processed digits $\begin{pmatrix} n' \\ m' \\ r' \end{pmatrix}$ which are the only part of the W-input that gets tested. Moreover, it is only tested whether $\begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. This is the situation that arises whenever the previously processed pair of digits adds up to 10 or more. In summary, the function w processes just two pieces of information:

(1) **3 digits:** w inputs $\binom{n}{m}$ from A and outputs one of the triples $\begin{pmatrix} n \\ m \\ n+m \end{pmatrix}$, or $\begin{pmatrix} n \\ m \\ n+m-9 \end{pmatrix}$, $\begin{pmatrix} n \\ m \\ n+m+1 \end{pmatrix}$, or $\begin{pmatrix} n \\ m \\ n+m-10 \end{pmatrix}$ in W;

(2) **the carry-over:** w tests whether its W-input begins with $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, and prepends $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ to its W-output whenever $n+m \geq 10$ if the test was negative, or $n+m+1 \geq 10$ if the test was positive.

Which one of the outputs in (1) is produced depends on the test in (2), and whether n and m incur a carry-over. The informations (1) and (2) are encoded in the two components of the product

$$\begin{aligned} V &= B \times Q \text{ where} \\ B &= \{0, 1, 2, \ldots, 9\} \text{ and} \\ Q &= \{0, 1\} \end{aligned}$$

The intended interpretation of $q \in Q$ is that

- $q = 0$ denotes the situation when $\begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, i.e. there is no carryover;
- $q = 1$ denotes the situation when $\begin{pmatrix} n' \\ m' \\ r' \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, i.e. there is carryover.

The simplified decimal adder is an instance of a schema in the form

$$\frac{\mathsf{dadd}_0 : A \to V \qquad \mathsf{dadd}_1 : A \times V \to V \qquad \mathsf{dadd}_2 : V \to B}{(\!|\mathsf{dadd}|\!)^\bullet : A^+ \to B} \tag{8}$$

On one hand, this schema is quite similar to the word induction in (2), extending it just by the dadd_2, which extracts the outputs, and leaves V as an internal space. On the other hand, (8) will turn out to be a *finite state machine*, over the state space V. State machines will be discussed in the next section. For the moment, the claim is that the finite set V, with 20 elements, suffices for the computation that originally required the infinite word space W. The functions implement addition in terms of (8) are

$$\mathsf{dadd}_0 \begin{pmatrix} n \\ m \end{pmatrix} = \begin{cases} \langle n+m, 0 \rangle & \text{if } n+m < 10 \\ \langle n+m-10, 1 \rangle & \text{if } n+m \geq 10 \end{cases}$$

$$\mathsf{dadd}_1 \left(\begin{pmatrix} n \\ m \end{pmatrix} , \langle r, q \rangle \right) = \begin{cases} \langle n+m, 0 \rangle & \text{if } q = 0 \ \wedge \ n+m < 10 \\ \langle n+m-10, 1 \rangle & \text{if } q = 0 \ \wedge \ n+m \geq 10 \\ \langle n+m+1, 0 \rangle & \text{if } q = 1 \ \wedge \ n+m+1 < 10 \\ \langle n+m-9, 1 \rangle & \text{if } q = 1 \ \wedge \ n+m+1 \geq 10 \end{cases}$$

$$\mathsf{dadd}_2 (r , q) = r$$

The idea is that the banana-function $(\!|\mathsf{dadd}|\!)^\bullet$ should be constructed by extending (3) along the function dadd_2, which extracts the output digits from W:

$$(\!|\mathsf{dadd}|\!) n = \mathsf{dadd}_0 (a) \qquad (\!|\mathsf{dadd}|\!) (n :: \boldsymbol{x}) = \mathsf{dadd}_1 (n, (\!|\mathsf{dadd}|\!) \boldsymbol{x}) \qquad (\!|\mathsf{dadd}|\!)^\bullet \boldsymbol{x} = \mathsf{dadd}_2 \circ (\!|\mathsf{dadd}|\!) \boldsymbol{x}$$

Running the decimal adder begins by reducing $(\!|\mathsf{dadd}|\!)$ to dadd_0 and dadd_1 by the same inductive descent like before:

$$\begin{aligned} (\!|\mathsf{dadd}|\!) \begin{pmatrix} 087 \\ 013 \end{pmatrix} &= (\!|\mathsf{dadd}|\!) \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} :: \begin{pmatrix} 8 \\ 1 \end{pmatrix} :: \begin{pmatrix} 7 \\ 3 \end{pmatrix} \right) = \\ &= \mathsf{dadd}_1 \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} , (\!|\mathsf{dadd}|\!) \left(\begin{pmatrix} 8 \\ 1 \end{pmatrix} :: \begin{pmatrix} 7 \\ 3 \end{pmatrix} \right) \right) = \\ &= \mathsf{dadd}_1 \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} , \mathsf{dadd}_1 \left(\begin{pmatrix} 8 \\ 1 \end{pmatrix} , (\!|\mathsf{dadd}|\!) \begin{pmatrix} 7 \\ 3 \end{pmatrix} \right) \right) = \\ &= \mathsf{dadd}_1 \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} , \mathsf{dadd}_1 \left(\begin{pmatrix} 8 \\ 1 \end{pmatrix} , \mathsf{dadd}_0 \begin{pmatrix} 7 \\ 3 \end{pmatrix} \right) \right) \end{aligned}$$

The outputs are produced by ascending back up through the composite of dadd_0 and dadd_1, while extracting the

outputs using dadd_2 along the way:

$$\mathsf{dadd}_0\begin{pmatrix}7\\3\end{pmatrix} = \langle 0,1\rangle \qquad\qquad \mathsf{dadd}_2(0,1) = 0$$

$$\mathsf{dadd}_1\left(\begin{pmatrix}8\\1\end{pmatrix}, \langle 0,1\rangle\right) = \langle 0,1\rangle \qquad\qquad \mathsf{dadd}_2(0,1) = 0$$

$$\mathsf{dadd}_1\left(\begin{pmatrix}0\\0\end{pmatrix}, \langle 0,1\rangle\right) = \langle 1,0\rangle \qquad\qquad \mathsf{dadd}_2(1,0) = 1$$

The outputs are produced one digit at a time, but the complete numeral can be accumulated by word induction again:

$$(\!|\mathsf{dadd}|\!)^{\#}a = (\!|\mathsf{dadd}|\!)^{\bullet}a \qquad\qquad (\!|\mathsf{dadd}|\!)^{\#}(a::\boldsymbol{x}) = \Big((\!|\mathsf{dadd}|\!)^{\bullet}(a::\boldsymbol{x})\Big) :: \Big((\!|\mathsf{dadd}|\!)^{\#}(\boldsymbol{x})\Big)$$

which gives

$$(\!|\mathsf{dadd}|\!)^{\#}\begin{pmatrix}087\\013\end{pmatrix} \quad = \quad 100$$

While this is much closer to children's addition algorithm, there is still one final wrinkle. Since $V = C \times Q$, the function dadd_1 is actually in the form $A \times C \times Q \to B \times Q$. But looking at its definition, we see that it does not depend on C.

The simple decimal adder is an instance of a schema in the form

$$\frac{\mathsf{dad}_0 : Q \qquad\qquad \mathsf{dad}_1 : A \times Q \to B \times Q}{(\!|\mathsf{dad}|\!) : A^{+} \to B} \tag{9}$$

where the initial state is $\mathsf{dad}_0 = 0$, i.e. there is no carry-over, and the transition function is

$$\mathsf{dad}_1\left(\begin{pmatrix}n\\m\end{pmatrix},\ q\right) \quad = \quad \begin{cases} \langle n+m, 0\rangle & \text{if } q = 0 \ \wedge\ n+m < 10 \\ \langle n+m-10, 1\rangle & \text{if } q = 0 \ \wedge\ n+m \geq 10 \\ \langle n+m+1, 0\rangle & \text{if } q = 1 \ \wedge\ n+m+1 < 10 \\ \langle n+m-9, 1\rangle & \text{if } q = 1 \ \wedge\ n+m+1 \geq 10 \end{cases}$$

The tabulated version of this function is in Fig. 5. The arrow on $q = 0$ marks that it is the initial state dad_0. The

dad_1	$q+n+m<10$	$q+n+m\geq 10$
$q=0\leftarrow$	$n+m,\ 0$	$n+m-10,\ 1$
$q=1$	$n+m+1,\ 0$	$n+m-9,\ 1$

Figure 5: The table of dad_1

corresponding state-transition diagram is in Fig. 6. The formal definitions and the execution model are spelled out in the next section.

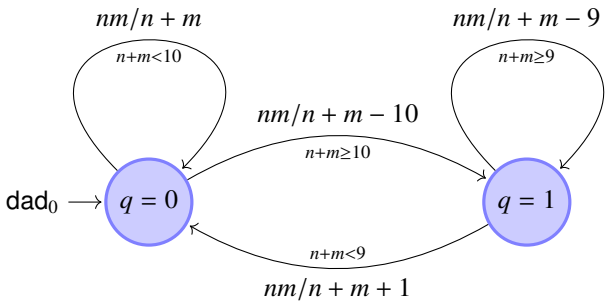


Figure 6: Simple decimal adder

2 Machines

Machines are built to interact with structured environments: cars interact with roads, printers print on precut sheets of paper, computers process suitably encoded data. In modeling computation, we usully talk about machines when the input and the output data are stored in some sort of *external* memory, and the machine interacts with that memory. Traditionally, the words processed are presented to a computer on tapes, subdivided into cells, so that each may carry a letter, like in the telegraphs that were in use when the computer was invented. In the current chapter, we study state machines that read and write their words sequentially. The tapes are thus spooled in one direction, and can be ignored. In the next chapter, the tapes will be spooled in both directions, which essentially changes the nature of the external memory.

State can be thought of as the *internal* memory, where the information relevant for the control flow of computation is stored, and the irrelevant information is filtered out. A memory update, i.e. a state change, is called a **transition**.

Definition 3. *A* transition (Mealy) machine *for inputs from A into outputs in B over the state space Q is the schema*

$$\frac{\rho_0 : V \qquad \rho_1 : Q \times A \to Q \times B}{(\!|\rho|\!)^\bullet : A^+ \to B} \tag{10}$$

The induced channel $(\!|\rho|\!)^\bullet$ *is defined*

$$(\!|\rho|\!)a = \rho_1\,(\rho_0, a) \qquad (\!|\rho|\!)\,(\boldsymbol{x}{::}a) = \rho_1\Big((\!|\rho|\!)^\circ \boldsymbol{x}, a\Big)$$
$$(\!|\rho|\!)^\circ \boldsymbol{x} = \pi^\circ \circ (\!|\rho|\!)\boldsymbol{x} \qquad (\!|\rho|\!)^\bullet \boldsymbol{x} = \pi^\bullet \circ (\!|\rho|\!)\boldsymbol{x} \tag{11}$$

where $\pi^\circ \colon Q \times B \to Q$ *and* $\pi^\bullet \colon Q \times B \to B$ *are the projections.*

Machines are functions extended in time. A transition function $\rho_1 \colon Q \times A \to Q \times B$ can be thought of as a function from A to B extended in time along the state space Q. It begins from the initial state $\rho_0 \colon Q$, and continues from state to state along $\rho_1{}^\circ \colon Q \times A \to Q$, as long as there are inputs from A. At each step, an output is produced by $\rho_1{}^\bullet \colon Q \times A \to B$. This describes the computation from (11). The output histories can be accummulated using

$$(\!|\rho|\!)^\# a = (\!|\rho|\!)^\bullet a \qquad (\!|\rho|\!)^\#(\boldsymbol{x}{::}a) = \Big((\!|\rho|\!)^\#(\boldsymbol{x})\Big) :: \Big((\!|\rho|\!)^\bullet(\boldsymbol{x}{::}a)\Big)$$

2.1 Tabulating finite transition machines

When there are finitely many states, a state machine boils down to a finite list of equations in the form $\rho_1(q, a) = \langle q', b\rangle$. The functions are tabulated by arranging the equations into matrices.

Example: Binary adder. The positional addition algorithm, taught for the decimal system at school, and formalized in Sec. 1, applies on all numeral bases. In the binary, the sum $87 + 13 = 100$ becomes $1010111 + 1101 = 1100100$. The task is now to describe a machine bad such that function

$$\begin{array}{ccccc} \{0,1\}^+ & \times & \{0,1\}^+ & \xrightarrow{(\!|\mathsf{bad}|\!)^\#} & \{0,1\}^+ \\ \langle 01010111, 00001101\rangle & & & \longmapsto & 01100100 \end{array} \tag{12}$$

where the numbers are padded to the same length, and extended by one additional leading 0 each, taking care for the carry-over from the last step. To implement this

algorithm on a machine, the input pair of bit words is zipped into an input word of bit pairs:

$$\begin{array}{ccccc} \{0,1\}^+ & \times & \{0,1\}^+ & \xrightarrow{\text{zip}} & \big(\{0,1\}\times\{0,1\}\big)^+ \\ \langle 01010111, 00001101\rangle & & & \longmapsto & 00\ \ 10\ \ 00\ \ 10\ \ 01\ \ 11\ \ 10\ \ 11 \end{array} \tag{13}$$

so that the addition task (12) becomes

$$\begin{array}{ccccc} (\{0,1\} & \times & \{0,1\})^+ & \xrightarrow{(\!|\mathsf{bad}|\!)^\#} & \{0,1\}^+ \\ 00\ \ 10\ \ 00\ \ 10\ \ 01\ \ 11\ \ 10\ \ 11 & & & \longmapsto & 01100100 \end{array} \tag{14}$$

The machine bad should thus have the same carry-nocarry state space Q like the decimal adder in Fig. 6, but now with the binary A-inputs and B-outputs, i.e.

$$A \;=\; \{0,1\}\times\{0,1\} \qquad\qquad B \;=\; \{0,1\} \;=\; Q$$

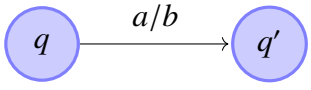
Since the numerals are written right to left, the transition function is in the form $\mathsf{bad}_1 : A \times Q \to B \times Q$, like in Sec. 1.5, and dually to Def. 3. The outputs in the form $\langle b, q\rangle$ are tabulated in Fig. 7.

bad_1	00	01	10	11
$q = 0 \leftarrow$	$0, 0$	$1, 0$	$1, 0$	$0, 1$
$q = 1$	$1, 0$	$0, 1$	$0, 1$	$1, 1$

Figure 7: The table of the binary adder. The arrow denotes the initial state bad_0.

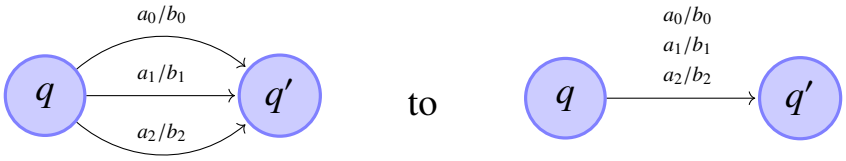
2.2 Drawing finite transition machines

The state-transition diagrams arise when the state machine equations $\rho_1(q, a) = \langle q', b\rangle$ are drawn as transitions:

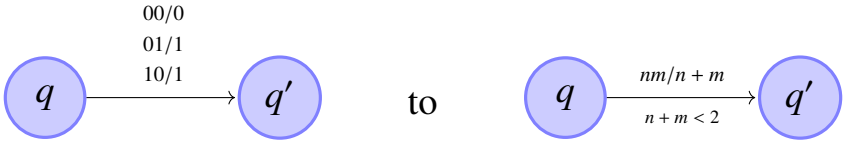


The directed graph comprised all such transitions is its transition diagram of the given machine.

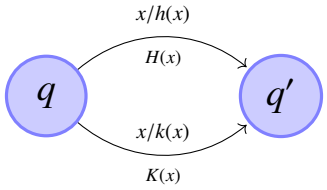
Notational conventions. When there are several edges between the same states, we list them over the same arrow:



This convention is used in Fig. 8 below. It decreases the number of edges in a diagram, but increases the number of labels on an edge. To simplify the labels, we use variables, and constrain them with a condition. E.g., we abbreviate



This convention was used in the decimal adder in Fig. 6. In general, for a family of transitions $\rho_1(q,x) = \langle q', h(x)\rangle$ for x satisfying $H(x)$ and $\rho_1(q,x) = \langle q', k(x)\rangle$ for x satisfying $K(x)$, we write



Back to the binary adder. Collecting the transitions of listed in Fig. **??** gives the diagram in Fig. 8.

2.3 Running transition machines

A run of the machine bad in Fig. 8 is displayed in Fig. 9. The machine starts at the state 0, *nocarry*, and reads the input 11. The state diagram shows that the

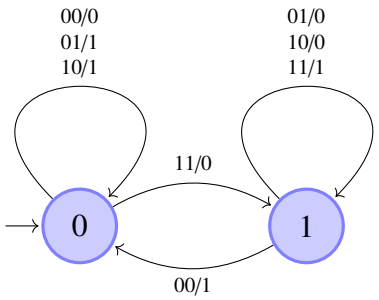


Figure 8: The transition diagram of the binary adder

machine should output 0 and transition to the state 1, *carry*. Indeed, 1+1 is 10 in the binary, and the machine outputs the second digit and carries the first digit as its state. After step 1, the machine is thus at the state 1, and ready to read the input 10. The state diagram shows the transition to the same state, and the output 0. That is step 2. Now we have two 0s at the output on the right, and the machine still carries 1 in its state, and is ready to read 11. And so on.[2]

Dynamics of machine computation. The binary adder bad implements the function $(\!|\mathsf{bad}|\!)^{\#}$ required by (14). Note, however, that this function does not take an input from $\{00,01,10,11\}^+$ to produce an output in $\{0,1\}^+$ at once, but incrementally

$$\{0,1\} \times \{00,01,10,11\} \xrightarrow{\mathsf{bad}_1} \{0,1\} \times \{0,1\}$$

$$\begin{array}{rcl} \{00,01,10,11\}^+ & \xrightarrow{(\!|\mathsf{bad}|\!)^{\bullet}} & \{0,1\} \\ (11) & \longmapsto & 0 \\ (10\ 11) & \longmapsto & 0 \\ (11\ 10\ 11) & \longmapsto & 1 \\ (01\ 11\ 10\ 11) & \longmapsto & 0 \\ (10\ 01\ 11\ 10\ 11) & \longmapsto & 0 \\ (00\ 10\ 01\ 11\ 10\ 11) & \longmapsto & 1 \\ (10\ 00\ 10\ 01\ 11\ 10\ 11) & \longmapsto & 1 \end{array}$$

The function $(\!|\mathsf{bad}|\!)^{\bullet}$, and its cummulative version s $(\!|\mathsf{bad}|\!)^{\#}$, are *dynamic*, i.e. *extended in time*, in the sense that the past inputs on the left determine the current *states* of the machine, which then influence the outputs. E.g., the input 10 is read at step 2, step 5 and step 7, where it halts. The first two times the output is 0, but the third time the output is 1. The difference is, of course, that the preceding inputs brought the machines before steps 2 and 5 to the state 1, whereas at the final step, the machine was at the state 0.

The general execution model whereby every a state machine with a chosen initial state, viz. a process, implements a mapping on data, like the machine add_2 above implements the addition, is called *coinduction*. We spell it out in the next section.

[2] Since there is no carry-over in the end, we omit extending both numbers by a leading 0.

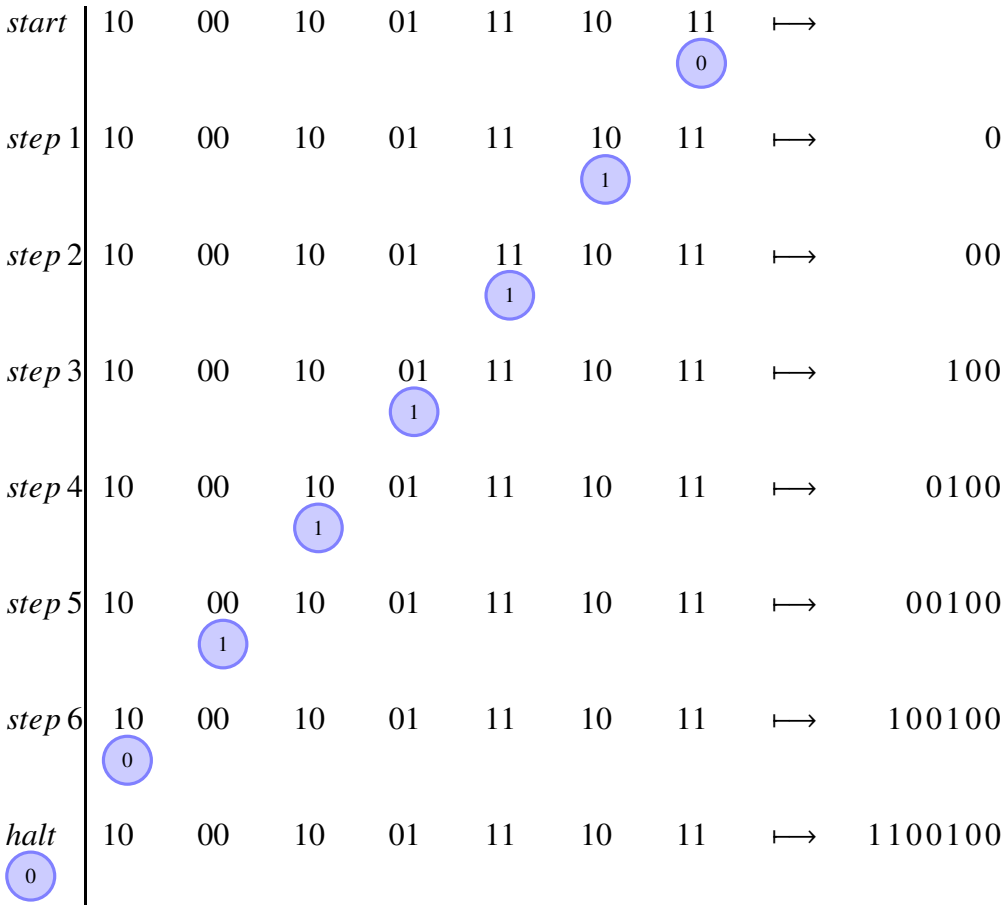


Figure 9: A run the binary adder in Fig. 8 computing 1010111 + 1101 in the form (14)

2.4 State machines

Definition 4. *A* state machine *for inputs from A into outputs in B over the state space V is the schema*

$$\frac{\rho_0 : A \to V \qquad \rho_1 : V \times A \to V \qquad \rho_2 : V \to B}{(\!|\rho|\!)^\bullet : A^+ \to B} \tag{15}$$

The induced channel $(\!|\rho|\!)^\bullet$ *is defined*

$$(\!|\rho|\!)^\circ a = \rho_0\,(a) \qquad (\!|\rho|\!)^\circ\,(\boldsymbol{x}{::}a) = \rho_1\Big((\!|\rho|\!)^\circ\,(\boldsymbol{x})\,, a\Big) \qquad (\!|\rho|\!)^\bullet \boldsymbol{x} = \rho_2 \circ (\!|\rho|\!)^\circ \boldsymbol{x} \tag{16}$$

Example: Binary adder as a state machine. The transition function is now in the form $\mathsf{badd}_1 : A \times V \to V$ for the types

$$A \;=\; \{0,1\} \times \{0,1\} \;=\; V \qquad\qquad B \;=\; \{0,1\}$$

and the table is in Fig. 10. The colors are used to relate it with Fig. **??**. If the state space can is interpreted as $V = B \times Q$, then $\mathsf{badd}_2 : V \to B$ projects the state bq to b.

badd_1	00	01	10	11	badd_2
$q = 00$	00	10	10	01	0
$q = 01$	10	01	01	11	0
$q = 10$	00	10	10	01	1
$q = 11$	10	01	01	11	1
badd_0	00	10	10	01	

Figure 10: The table of the binary adder state machine

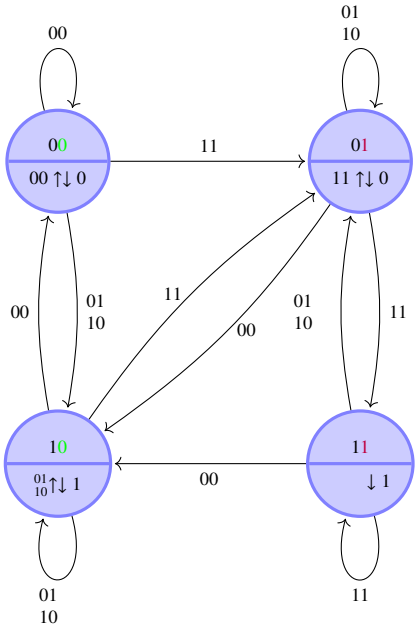


Figure 11: The state diagram of the binary adder

Proposition 5. *For any pair of types A and B, the (Moore) state machines and the (Mealy) transition machines implement the same channels* $A^+ \to B$.

Proof (Sketch). A state machine over a state space V induces a transition machine over the state space $Q = 1 + V$, i.e. with one

additional state:

$$\frac{\rho_0 \colon A \to V \qquad \rho_1 \colon V \times A \to V \qquad \rho_2 \colon V \to B}{\widetilde{\rho}_1 = \Big((1+V)\times A \cong A + (V \times A) \xrightarrow{[\rho_0,\rho_1]} V \xrightarrow{\langle \iota,\rho_2\rangle} (1+V)\times B\Big)}$$

where $V \xrightarrow{\iota} 1 + V$ is the inclusion. Taking the initial state $\widetilde{\rho}_0 : 1 + V$ to be the element of 1, it is easy to prove that $(\!|\widetilde{\rho}|\!)^\bullet = (\!|\rho|\!)^\bullet$.

The other way around, given a transition machine ρ over a state space Q, let $V = Q \times B$ and derive the state machine with

$$\frac{\rho_0 \colon Q \qquad \rho_1 \colon Q \times A \to Q \times B}{\widehat{\rho}_0 = \Big(A \cong 1 \times A \xrightarrow{\rho_0 \times A} Q \times A \xrightarrow{\rho_1} Q \times B = V\Big)}$$

Keeping $\widehat{\rho}_1 = \Big(V \times A \xrightarrow{\pi \times A} Q \times A \xrightarrow{\rho_1} Q \times B = V\Big)$ and $\widehat{\rho}_2 = \Big(V = Q \times B \xrightarrow{\pi} B\Big)$ gives $(\!|\widehat{\rho}|\!)^\bullet = (\!|\rho|\!)^\bullet$ again. □

For completeness, we mention the version of state machine which combines Definitions 3 and 4, and implements channels that input lists, including the empty one, and not just words.

Definition 6. *A* Moore machine *for inputs from A into outputs in B over the state space V is the schema*

$$\frac{\rho_0 : V \qquad \rho_1 \colon V \times A \to V \qquad \rho_2 \colon V \to B}{(\!|\rho|\!)^\bullet \colon A^* \to B} \tag{17}$$

The induced channel $(\!|\rho|\!)^\bullet$ *is defined*

$$(\!|\rho|\!)^\circ() = \rho_0 \qquad (\!|\rho|\!)^\circ(\boldsymbol{x}{::}a) = \rho_1\Big((\!|\rho|\!)^\circ(\boldsymbol{x}), a\Big) \qquad (\!|\rho|\!)^\bullet \boldsymbol{x} = \rho_2 \circ (\!|\rho|\!)\boldsymbol{x} \tag{18}$$

A Moore machine produces its first output as soon as it enters the initial state, i.e. before it consumes its first input. To perform the addition, it would need to be provided an additional, *blank* symbol for this purpose, and an initial state where this symbol is output. But while the Moore machine conventions are inconvenient for addition, they are convenient for other purposes, as some of the exercises illustrate.

2.5 Exercises

a) Use tables and diagrams to design machines that compute the following functions

   i. addition in the base 3

   ii. addition in the base 8

   iii. multiplication with 2

   iv. binary successor

b) Consider the transition machine bod over the input and output types $A = B = \{0, 1\}$ and the state space $Q = \{00, 01, 10, 11\}$, with the initial state $\mathsf{bod}_0 = 00$, and the the transition function $\mathsf{bod}_1 : Q \times A \to Q \times B$ tabulated by

bod_1	0	1
→ 00	01,0	10, 0
01	00,0	11, 0
10	11,0	00, 0
11	10,1	01, 1

   Draw the diagram and determine the banana-function $(\!|\mathsf{bod}|\!)^\bullet \colon \{0,1\}^+ \to \{0,1\}$.

c) Tabulate and determine the banana function of the Moore machine displayed in Fig. 12.

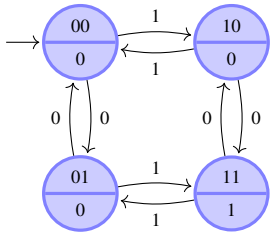


Figure 12: The Moore machine for Ex. (c)

d) Implement the function Mod_3 that inputs the binary representation of a non-negative integer and outputs the residue of its division with 3:

$$\begin{array}{ccc} \{0,1\}^+ & \xrightarrow{Mod_3} & \{0,1,2\} \\ \boldsymbol{a} & \mapsto & \overline{\boldsymbol{a}} \mod 3 \end{array}$$

where $\boldsymbol{a} = (a_{n-1} \cdots a_1\ a_0)$, $\overline{\boldsymbol{a}} = \sum_{i=0}^{n} a_i 2^i$, and $b = a \mod 3$ and $b = a \mod 3$ is the smallest non-negative integer such that there is $k \in \mathbb{N}$ such that $a = b + 3k$.

3 Automata

Automata are machines where the outputs are of type $B = 2 = \{0, 1\}$. The implemented functions $A^+ \to 2$ or $A^* \to 2$ can be viewed as characterisitic funtions, characterizing sets of words or lists. Def. 1 stipulated that a set of words or lisrs is called a language.

Definition 7. *An* automaton[3] *over an alphabet A and a state space V is the schema*

$$\frac{\rho_0 : V \qquad \rho_1 : V \times A \to V \qquad \rho_2 \subseteq V}{\mathcal{L}_\rho \subseteq A^*} \tag{19}$$

The elements of ρ_2 are called the accepting states*, and the set $\mathcal{L}_\rho$ is the* accepted language *of the automaton. The banana-function $(\!|\rho|\!)^\bullet : A^* \to V$ determines the accepted language as the set of words that take it to any of the accepting states, i.e.*

$$\begin{array}{rcl} \mathcal{L}_\rho & = & \{\boldsymbol{x} \in A^* \mid (\!|\rho|\!)^\circ \boldsymbol{x} \in \rho_2\} \\ & & \textit{where} \quad (\!|\rho|\!)^\circ () = \rho_0 \quad \textit{and} \quad (\!|\rho|\!)^\circ (\boldsymbol{x}::a) = \rho_1\Big((\!|\rho|\!)^\circ (\boldsymbol{x}), a\Big) \end{array} \tag{20}$$

Drawing automata. If the only outputs of a Moore machine are 0 or 1, then it is not necessary to explicitly mark the outputs at all states, but only at those that output one of the symbols, and leave the rest unmarked. The convention is to mark the accepting states, where the output would be 1, by a double circle. The remaining states are assumed to be rejecting, i.e. output 0. The automaton diagram corresponding to the Moore machine in Fig. 12 is in Fig. 13.

[3]The plural of the word *"automaton"* is *"automata"*.

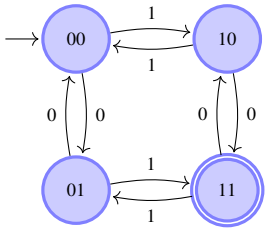


Figure 13: The automaton accepting the binary words with odd numbers of 0s and 1s

3.1 Where do machines produce their outputs?

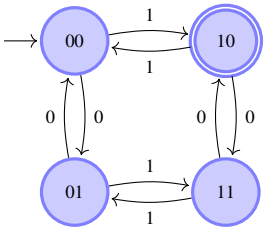
Summarizing how the different kinds of machines compute, note that

- transition machines defined in 3 produce the output letters at transitions, where they also consume the input letters,
- state machines produce the output letters at states, but
  - a state machine in 4 reaches a state *after* it consumes an input letter;
  - a Moore machine in **??** produces an output *before* it consumes an input,
- automata do not produce outputs, but accept or reject the input words.

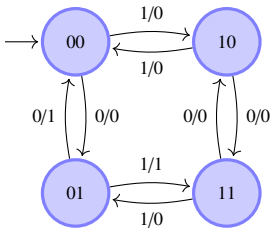
3.2 Exercises

Determine

a) the accepted language of the following automaton:



b) the behavior of the following machine:



4 Coalgebras and coinduction

4.1 Stream machines capture induction

A stream machine is a machine where the input alphabet is the unit type, or the singleton set $A = 1$. An input word from this alphabet conveys its length as the only information. The inputs that a stream machine may receive can be viewed as natural numbers

$$1^* \;=\; \left\{ (), (0), (00), \ldots, (\overbrace{00\cdots0}^{n}), \ldots \right\} \;=\; \cong \;\; \{0, 1, 2, \ldots, n, \ldots\} \;=\; \mathbb{N}$$

A stream $f = \left(f_0, f_1, f_2, \ldots, f_n, \ldots \right)$ of data from B can be viewed as a function $f : 1^* \to B$. Note that the the sets of lists 1^* and of words 1^+ are in one-to-one correspondence, given by adding and dropping the unique letter of the alphabet. The difference between the transition machines and the state machines disappears. Counting re-emerges as the arithmetic in base 1, and the induction re-emerges as a special case of the machine computing.

Proposition 8. *Compare the inductive schema from Def.* **??** *and the stream machine, viewed as the special case of Def. 3 for $A = 1$. The two schemas are aligned here:*

$$\frac{b\colon Q \qquad\qquad q\colon Q \to Q}{(\!|b, q|\!)\colon \mathbb{N} \to Q} \qquad\qquad \frac{\rho_0\colon Q \qquad\qquad \rho_1\colon Q \to B \times Q}{(\!|\rho|\!)\colon \mathbb{N} \to B \times Q}$$

The input words from $A^+ = 1^+$ are identified with their lengths in $\mathbb{N}$, and $(\!|\rho|\!)$ is the pair $\langle (\!|\rho|\!)^\bullet, (\!|\rho|\!)^\circ \rangle$, written in the notation of Def. 3. The two schemas are equivalent modulo the output map $\rho_1^\bullet : Q \to B$, in the following sense:

- *The induction schema is the special case of the stream coinduction for $B = 1$ and*

$$(\!|b, q|\!) \;=\; (\!|\rho|\!) \qquad \text{for } \rho_0 = b \text{ and } \rho_1 = q \tag{21}$$

- *The stream coinduction boils down to induction and $\rho_1^\bullet$ in the form*

$$(\!|\rho|\!) \;=\; \left\langle (\!|b, q|\!) \;,\; \rho_1^\bullet \circ (\!|b, q|\!) \right\rangle \qquad \text{for } b = \rho_0 \text{ and } q = \rho_1^\circ$$

where $\rho_1^\circ = \left(Q \xrightarrow{\rho_1} B \times Q \xrightarrow{\pi^\circ} Q \right)$ and $\rho_1^\bullet = \left(Q \xrightarrow{\rho_1} B \times Q \xrightarrow{\pi^\bullet} B \right)$.

The proof is straightforward. With some additional programming, the correspondence can be extended to the recursion schema.

Left-to-right *vs* right-to-left again. In Sec. 1, the fact that many languages are written left-to-right, but most numerals are right-to-left, required that we keep switching between writing dual conventions of writing the transition functions $V \times A \to V \times B$ and $A \times V \to B \times V$. Streams are written from the left, leaving their infinite tails to extend to the right. But this means that the further computational steps can must extend a stream before its beginning on the left, and not on its infinite side on the right. The transition functions are therefore in the form $A \times V \to B \times V$, like for writing right-to-left.

4.2 Stream abstraction

For any given transition function $\rho\colon Q \to B \times Q$, choosing an initial state $x : Q$ induces a B-stream. Since x can be any state, this induces a function from states to streams.

$$\frac{x\colon Q \qquad \rho\colon Q \to B \times Q}{(\!|x,\rho|\!)\colon \mathbb{N} \to B} \qquad \rightsquigarrow \qquad \frac{\dfrac{\rho\colon Q \to B \times Q}{(\!|-,\rho|\!)\colon Q \times \mathbb{N} \to B\colon \big(\langle x, n\rangle \mapsto (\!|x,\rho|\!)_n\big)}}{[\![\rho]\!] : Q \to B^{\mathbb{N}}\colon \big(x \mapsto \lambda n.\ (\!|x,\rho|\!)_n\big)} \tag{22}$$

Here $B^{\mathbb{N}}$ denotes set of streams $\mathbb{N} \to B$, and the last step is the function abstraction, discussed in Sec. **??**. By abstraction, the family of banana-functions $(\!|x,\rho|\!) : \mathbb{N} \to B$, indexed over $x \in Q$ induces the *lens* function $[\![\rho]\!] : Q \to B^{\mathbb{N}}$. Every state x thus induces a B-stream

$$[\![\rho]\!]\, x \quad = \quad \big((\!|x,\rho|\!)_0,\ (\!|x,\rho|\!)_1,\ (\!|x,\rho|\!)_2, \ldots, (\!|x,\rho|\!)_n, \ldots \big)$$

Every stream in $B^{\mathbb{N}}$, on the other hand, decomposes into a *head* in B and a *tail* in $B^{\mathbb{N}}$, which together give

$$\begin{array}{rcl} B^{\mathbb{N}} & \xrightarrow{\langle \mathsf{h},\mathsf{t}\rangle} & B \times B^{\mathbb{N}} \\ (f_0,\ f_1,\ f_2, \ldots, f_n, \ldots) & \mapsto & \big\langle f_0, (f_1, f_2, \ldots, f_n, \ldots)\big\rangle \end{array} \tag{23}$$

Noticing that $\mathsf{h}\,([\![\rho]\!]\, x)$ is $\rho^\bullet x$, whereas $\mathsf{t}\,([\![\rho]\!]\, x)$ reduces to $[\![\rho]\!]\,(\rho^\circ x)$ means that diagram in Fig. 15 commutes.

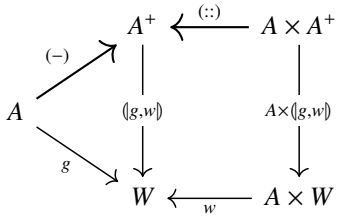


Figure 14: A banana-function from the initial A-word algebra

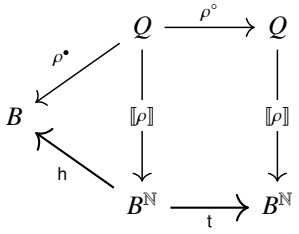


Figure 15: A lens-function to the final B-stream coalgebra

B-stream coalgebras are the functions in the form $Q \to B \times Q$. They are the transition functions of some stream machines, but now they acquire an independent role, as machines' initial states $x : Q$ have been abstracted away in (22). Different forms of stateful behaviors are studied using different varieties of coalgebras.

Lens-functions $[\![\rho]\!] : Q \to B^{\mathbb{N}}$ represent the states of arbitrary B-stream coalgebras as B-streams. Formally, this representation is the B-stream coalgebra structure preservation. The B-stream coalgebra structures the functions $\rho : Q \to B \times Q$ and $\langle \mathsf{h}, \mathsf{t}\rangle : B^{\mathbb{N}} \to B \times B^{\mathbb{N}}$, displayed in Fig. 15. The fact that $[\![\rho]\!]$ preserves this structure is witnessed by the fact that the diagram in Fig. 15 commutes. The fact that every B-stream coalgebra induces a unique lens-function to $\langle \mathsf{h}, \mathsf{t}\rangle : B^{\mathbb{N}} \to B \times B^{\mathbb{N}}$ means that $B^{\mathbb{N}}$ carries the structure of a *final* B-stream coalgebra.

4.3 Channel abstraction

We call ***AB*-machine coalgebras** the functions in the form $Q \times A \to Q \times B$ [4]. They are the transition functions of transition machines. In this section they take the center stage, since we abstract over the initial states $x : Q$ again:

$$\frac{x\colon Q \qquad \rho\colon A \times Q \to B \times Q}{(\!|x,\rho|\!)\colon A^+ \to B \times Q} \qquad \rightsquigarrow \qquad \frac{\dfrac{\rho\colon A \times Q \to B \times Q}{(\!|-,\rho|\!)\colon Q \times A^+ \to B\colon \big(\langle x, \boldsymbol{a}\rangle \mapsto (\!|x,\rho|\!)\boldsymbol{a}\big)}}{[\![\rho]\!] : Q \to B^{A^+}\colon \big(x \mapsto \lambda \boldsymbol{a}.\ (\!|x,\rho|\!)\boldsymbol{a}\big)} \tag{24}$$

Here B^{A^+} denotes set of channels $A^+ \to B$. The family of banana-functions $(\!|x,\rho|\!) : A^+ \to B$, indexed over $x : Q$ thus induces the *lens* function $[\![\rho]\!] : Q \to B^{A^+}$. The derivation in (24) induces the lens-function in Fig. 16. The *AB*-

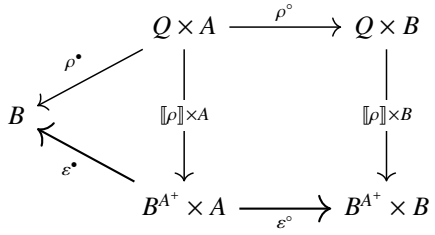


Figure 16: A lens-function from an *AB*-machine coalgebra to the final *AB*-machine coalgebra of *AB*-channels

channels in B^{A^+} thus form the state space of the final *AB*-machine coalgebra. The coalgebra structure on B^{A^+} is the pair $\varepsilon = \langle \varepsilon^\circ, \varepsilon^\bullet \rangle$ where $\varepsilon^\bullet(\varphi, a) = \varphi(a)$ is the evaluation, whereas $\varepsilon^\circ(\varphi, a) = \varphi^{::a}$ is the prefixing. More precisely,

$$\begin{array}{ccccc} B^{A^+} \times A & \xrightarrow{\varepsilon=\langle \varepsilon^\circ, \varepsilon^\bullet\rangle} & B^{A^+} \times B & \qquad \text{where} \qquad & A^+ \xrightarrow{\varphi^{::a}} B \\ \langle \varphi,\ a\rangle & \mapsto & \langle \varphi^{::a},\ \varphi(a)\rangle & & \boldsymbol{x} \mapsto \varphi(\boldsymbol{x}{::}a) \end{array} \tag{24}$$

The channel abstraction schema is the derivation

$$\frac{\rho\colon Q \times A \longrightarrow Q \times B}{[\![\rho]\!] : Q \longrightarrow B^{A^+}} \tag{25}$$

The lens-function $[\![\rho]\!]$, mapping each *AB*-machine state $q : Q$ into the induced channel $[\![\rho]\!]_q : B^{A^+}$ is defined inductively:

$$[\![\rho]\!]_q(a) \ = \ \rho^\bullet(q, a) \qquad\qquad [\![\rho]\!]_q(\boldsymbol{x}{::}a) \ = \ [\![\rho]\!]_{\rho^\circ(q,a)}(\boldsymbol{x}) \tag{26}$$

Its coalgebra preservation properties are in Fig. 17.

Channel abstraction realizes the coalgebraic semantics of machines. The coinductive interpretation of an *AB*-machine ρ assigns it a computational meaning. If machines are thought of as programs, then their channel abstractions (25) their semantics. This is a germ of the concept that will be studied in Ch. **??**.

[4] also with the inputs and the outputs written on the left

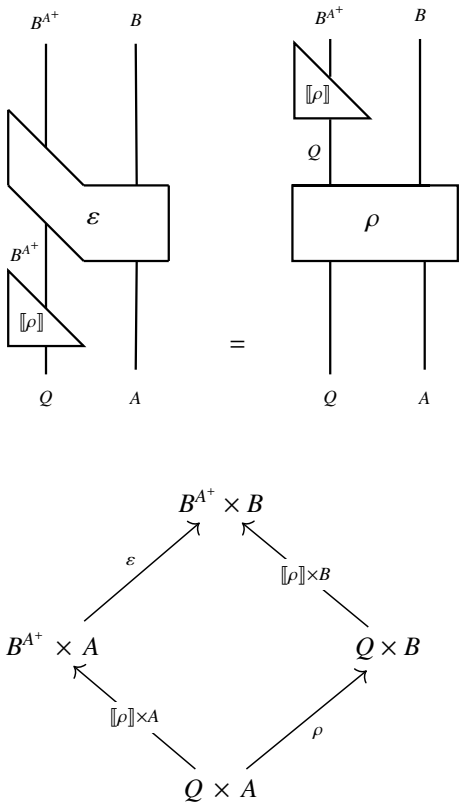
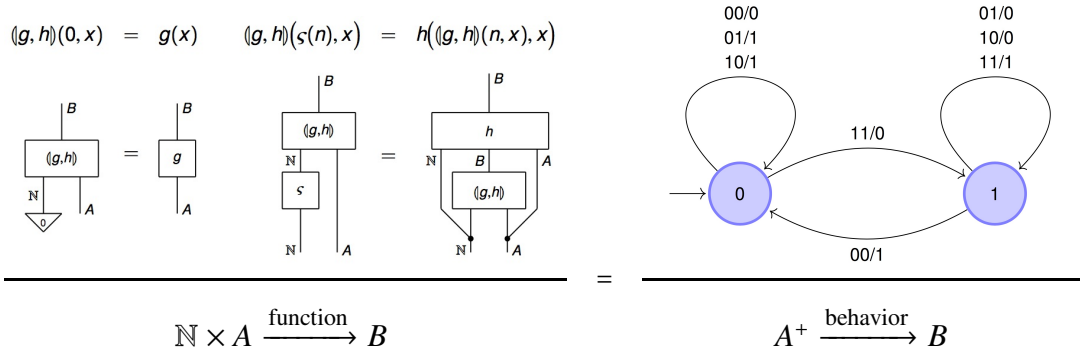


Figure 17: The AB-channel abstraction $[\![\rho]\!]$ preserves the AB-machine coalgebra structure

5 Summary, and open ends

5.1 What did we study?

Machines implement behaviors. In Chapter **??**, we learned how to specify sequences of *functions* using the induction schema. In the current Chapter, we learned how to specify *behaviors* using the coinduction schema. Just like the inductive and recursive specifications allowed computing functions, machines allow computing behaviors:



Behaviors $A^+ \longrightarrow B$ can be thought of as functions with memory: the output depends on whole histories of inputs. Machines implement behaviors by providing *bounded* memory as state.

States provide bounded memory. A state machine $Q \times A \xrightarrow{\chi} Q \times B$ consists of two maps

$Q \times A \xrightarrow{\chi_1} B$ maps inputs from A to outputs in B depending on the states in Q;

$Q \times A \xrightarrow{\chi_0} Q$ updates the state in Q depending on the input from A.

Since the state changes depend on the inputs, the states record some properties of the past (e.g., whether a past overflow is still carried or not). That is the sense in which states provide memory.

Implemented by state machines and thus equipped with memory, behaviors are past-sensitive functions.

Behaviors provide a universal representation of states. Given a machine $Q \times A \xrightarrow{\chi} Q \times B$, a choice of the initial state $s \in Q$ determines a behavior $A^+ \to B$. This behavior tells what outputs will be computed for all possible input histories. The process $\langle s, \chi \rangle$ is thus represented by the behavior $[\![\chi]\!]_s$ that it induces by coinduction. The behavior machine $B^{A^+} \times A \xrightarrow{\varepsilon} B^{A^+} \times B$ thus contains representations of all AB-processes. It is thus *universal*.

Upshot. The crucial feature of coinduction is that it supports the *universal* representation of computational behaviors.

Induction, and recursion as its parametrized version, built computation upon counting: e.g., the addition was defined as iterated counting, multiplication as iterated addition, etc. Using the projection, we were able to count backwards, and define the prececessor function, from that we derived still other functions.

Coinduction brings us beyond counting. At the beginning of this section, we saw how the basic concept of *state* makes it possible to add m and n without counting up to $m + n$, as it must be done when they are added recursively.

The crucial final step was use behaviors (or streams, or languages) as universal states, providing the universal representation of all states, assigning the identical representatives those that induce the same behaviors. This representation thus provides a *behavioral semantics* of state machines.

However, this behavioral semantics requires infinitely many states-as-behaviors. Even if all machines that we design have finitely many states, their size is unbounded, and the universal machine where they all embed must be infinite. They all have finite memories, but all together they require an infinite memory.

The crucial insight, arising from the work of Alan Turing, was that all computational behaviors, infinitely many of them, can be represented using a *finite* universal machine — provided that we give it an unbounded *external* memory, implemented as a "tape", and not as an internal state. This led to the concept of *programming*, which we will explore in the next section.

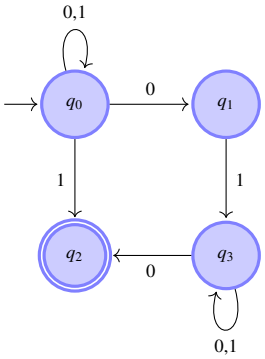
But before we go there, it may be interesting to mention what we are skipping.

5.2 What did we not study?

We studied machines in the form $Q \times A \to Q \times B$, which for the special case $A = B = \{0, 1\}$ boils down to automata in the form $Q \times \{0, 1\} \to Q$, with finite states $\Phi \subseteq Q$.

Nondeterminism. The following automaton is in the form $Q \times \{0, 1\} \to \wp Q$, where $\wp X = \{V \subseteq X\}$ denotes the set of

the subsets of X.



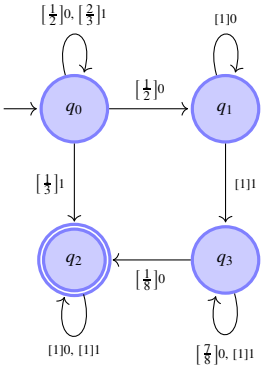
	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{\}$	$\{q_3\}$
$\leftarrow q_2$	$\{\}$	$\{\}$
q_3	$\{q_2, q_3\}$	$\{q_3\}$

(27)

Entering 0 at the initial state q_0 can have two different results: the machine can stay at q_0, or transition to q_1. Similarly at q_3: entering 0 can lead to staying at q_3, or to transitioning to q_2. On the other hand, entering 0 at q_1 leads to no results: the machine is stuck. At q_2, the machine is stuck for any input.

The above machine is *nondeterministic*. We did not study such machines. Machines and automata that we did study were *deterministic*, in the sense that at each state each input determined a unique state and output, mapped by the deterministic transition function $Q \times A \to Q \times B$. The nondeterministic transition function is in the form $Q \times A \to \wp(Q \times B)$, which means that the machine transition to multiple states and outputs, or to none, in which case it deadlocks. So it is not always determined what the machine will do. This is often used to account for the influences of the environment, or for some hidden variables not accounted in the design. Nevertheless, it is not hard to see that nondeteministic automata accept the same languages like the determinstc ones, although the nondeterministic computations may run exponentially faster, or require exponentially fewer states, or less memory. Nondeterministic machines in the form $Q \times A \to Q \times \wp B$ implement behaviors in the form $A^+ \to \wp B$. Nondeterministic machines in the general form $Q \times A \to \wp(Q \times B)$ implement $A \times B$-labelled irredundant trees [**?**] as their behavior, or the corresponding non-wellfounded sets [**?**].

Randomization. The idea of nondeterminism can be further refined by specifying not just several *possible* outcomes of a computational step, but by also assigning to them the *probabilities* with which each of the outcomes may occur. With this interpretation in mind, consider the following automaton.



	0	1
$\rightarrow q_0$	$\frac{1}{2}q_0 + \frac{1}{2}q_1$	$\frac{2}{3}q_0 + \frac{1}{3}q_2$
q_1	q_1	q_3
$\leftarrow q_2$	q_2	q_2
q_3	$\frac{1}{8}q_2 + \frac{7}{8}q_3$	q_3

(28)

Entering 0 at the initial state q_0 can still have the same two results as before: the machine can stay at q_0, or transition to q_1; but now we also know that the chance that either of these two results will occur is the same: exactly $\frac{1}{2}$. This means that running the machine 100 times, and each time entering 0 at the state q_0 will cause the machine to transition

to q_1 in roughly 50 cases, whereas it will leave it at the state q_0 in approximately the same number of cases. On the other hand, entering 1 at q_0 will lead to q_2 in $\frac{1}{3}$ of the cases, and leave the machine in the state q_0 in $\frac{2}{3}$ of the cases. Entering 0 at q_3 will lead to q_2 in roughly $\frac{1}{8}$ of the cases, and leave the machine in the state q_3 in the remaining $\frac{7}{8}$ of the cases. Entering 1 at q_3 will always leave the machine in the same state. And so on.

While nondeterministic machines were formalized using the powerset constructor $\wp X = \{V \subseteq X\}$, which could be equivalently presented in the form

$$\wp X \quad = \quad \{\gamma_V : X \to \{0, 1\}\}$$

by setting $\gamma_V(x) = 1$ if $x \in V$, otherwise $\gamma_V(x) = 0$, randomized machines can be formalized using the constructor

$$\mathcal{D}X \quad = \quad \{\mu_f : X \to [0, 1] \,\big|\, \sum_{x \in X} \mu(x) = 1\}$$

While the characteristic functions $\gamma : X \to \{0, 1\}$ only take the values 0 or 1, the probability distributions $\mu : X \to [0, 1]$ may also take the values in-between 0 and 1. Moreover, the values of a probability distribution add up to 1, as they measure the probabilities as the relative sizes of the parts of the space of all possibilities, by counting or estimating the frequency with which each of the possible events occur. The subscript f in μ_f means that the probability distribution μ is *finitely supported*. This means that the set $\{x \in X \mid \mu(x) \neq 0\}$ of events that may occur is finite, and not empty. Obviously, when X is finite, then all probability distributions over it are finitely supported.

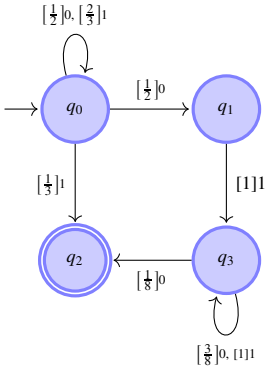
While the *possible* outcomes of a computational step in a *nondeterministic* machine were captured by the transition functions in the form

$$Q \times A \quad \to \quad \wp(Q \times B)$$

the *probable* outcomes in a *randomized* machine are captured by the transition functions in the form

$$Q \times A \quad \to \quad \mathcal{D}(Q \times B)$$

Since the set of the possible outcomes $\{x \in X \mid \mu_f(x) \neq 0\}$ is nonempty, a randomized machine never deadlocks: at every state, for every input there is some output. That is why the machine on (28) must have the transitions for 0 at q_1 and for 0 and 1 at q_2, at the states where the machine on (27) deadlocks. Randomized machines that can also deadlock, such as the automaton



	0	1
$\to q_0$	$\frac{1}{2}q_0 + \frac{1}{2}q_1$	$\frac{2}{3}q_0 + \frac{1}{3}q_2$
q_1	0	q_3
$\leftarrow q_2$	0	0
q_3	$\frac{1}{8}q_2 + \frac{3}{8}q_3$	q_3

(29)

are captured by the transition functions in the form $Q \times A \to \mathcal{D}_{\leq}(Q \times B)$, where the constructor

$$\mathcal{D}_{\leq}X \quad = \quad \{\mu_f : X \to [0,1] \,\big|\, \sum_{x \in X} \mu_f(x) \leq 1\}$$

collects *sub*probability distributions over X.

Exceptions and other side effects. Given a set E of computational effects, such as *exceptions* or *side effects* (e.g., deadlock, livelock, divergence, etc), then machines that implement these effects can be modeled in the form $Q \times A \to \mathcal{E}(Q \times B)$, where the constructor $\mathcal{E}X = E + X$ extends the set of the outcomes with the computational effects that are of interest. The general treatment of computational effects is in [**?**]. Its applications in functional programming are in [**?**, **?**].

5.3 What else can we compute?

We started this set of lectures by analyzing how we add numbers, and then implementing the positional addition of arbitrary pairs of binary (or decimal) numbers as state machines. In the meantime, we implemented many other algorithms as finite state machines, including the positional multiplication of arbitrary binary numbers with a fixed number. We finish the section by showing that the positional multiplication of arbitrary pairs of binary numbers cannot be implemented by a finite state machine. While we can compute many things by finite state machines, we cannot multiply arbitrary numbers. Why is that?

Suppose that there is a machine $mult_2$ with m states that multiplies binary numbers of arbitrary lengths. Take an arbitrary number $n > m$, and consider how would the machine $mult_2$ multiply the number 2^n with itself, to get $2^n \times 2^n = 2^{2n}$. Just like the positional binary addition, the positional binary multiplication would be a list processing algorithm, and the machine $mult_2$ would thus have to implement the function

$$\begin{array}{ccccc} \{0,1\}^* & \times & \{0,1\}^* & \xrightarrow{Mult_2} & \{0,1\}^* \\ \langle 1\underbrace{00\cdots0}_{n}, & & 1\underbrace{00\cdots0}_{n}\rangle & \longmapsto & 1\underbrace{00\cdots0}_{2n} \end{array} \tag{30}$$

where 1 with n 0s is the binary notation for 2^n. As explained in Sec. 2.3, the execution model of state machines requires that the multiple arguments that they may need to input be zipped into a single argument, as in (13), so the list processing to multiply binary numbers would actually have to be in the form

$$\begin{array}{ccc} (\{0,1\} \times \{0,1\})^* & \xrightarrow{Mult_2} & \{0,1\}^* \\ 11\ \underbrace{00\ 00\ \cdots\ 00}_{n} & \longmapsto & 11\ \underbrace{00\ 00\ \cdots\ 00}_{2n} \end{array} \tag{31}$$

This is the same as in the case of binary addition in Sec. **??**, where the function in the form (12) also needed to be implemented in the form (14). The difference is, though, that the input of length n induced the output of length $n+1$ in (14), whereas in (31) the input of length $n+1$ induces the output of length $2n+1$. Since a machine can only produce an output symbol when an input symbol is entered, the input in (31) would need to be padded with n additional symbols, in order to output $2n+1$ symbols at the output. The only way to pad binary numbers without changing their values is to add leading 0s on the left. The input in (31) would thus have to be padded with n pairs 00 on the left. After reading the $n+1$ significant pairs from the right, and writing the $n+1$ rightmost 0s as the output, the machine $mult_2$ would thus have to read the additional n pairs 00, that were added as the padding. These inputs should result at the output in

$n-1$ 0s and the rightmost 1. Since all n inputs are the same, this change in the outputs from 0 to 1 would have to arise from a change of state. But since we assumed that $n > m$, where m is the number of states, the machine $mult_2$ would have to revisit some state twice while reading the n pairs of 00s. So if the state where 1 is written on the output 00 has to be visited at the final step, then it also had to be visited at some of the n steps when the padding of n 00s was read. But then a 1 had to also be written earlier, and the output of the machine cannot be the binary representation of 2^{2n}, which consists of just one leading 1 and $2n$ 0s.

The conclusion is that the state machine $mult_2$ with m states cannot correctly multiply the number 2^n with itself, for $n > m$.

Intuitively, the reason why the positional addition can be implemented within the machine execution model from Sec. 2.3, whereas the positional multiplication cannot, is that the positional addition can be implemented using purely local operations on the digits, and the states only need to record whether there are carryovers from the previous digit additions or not; whereas the positional multiplication of a pair of numbers of length n requires at least n states, to record the position of the current local operation on the digits. A machine with bounded memory cannot multiply numbers of unbounded length. In the next lecture we shall learn that a machine with finite internal memory, and finite but unbounded, i.e. always extensible external memory can not only multiply numbers of unbounded length, but it can do substantially more. In fact, there is a machine that can do everything that all other machines can do.