The Myhill-Nerode Theorem

Equivalence Relations, Partitions

We start with a brief introduction of equivalence relations and partitions because these are central to the discussion of finite-state machines.

Informally, two objects are equivalent if distinguishing between them is of no interest. At the checkout counter of a supermarket, when you are getting 25 cents back, one quarter looks as good as the next. But the difference between a dime and a quarter is of interest. The set of all quarters is what mathematicians call an "equivalence class." The set of all dimes is another such class, as is the set of all pennies, and so on. The set of all coins gets carved up, or "partitioned," into disjoint subsets, as illustrated in Figure 2.12. Each subset is called an "equivalence class." Collectively, they form a "partition" of the set of all coins: every coin belongs to one and only one subset. There are finitely many such subsets in this example (6 to be precise).

Partitions ≡ Equivalence Relations

Given a partition, there is a unique equivalence relation that goes with it: the relation of belonging to the same subset.

And given an equivalence relation, there is a unique partition that goes with it. Draw the graph of the relation. It consists of disconnected complete subgraphs whose node sets form the sets of the partition.[3]

[3]Note: This is easier to visualize when the set of nodes is finite, but works also for infinite

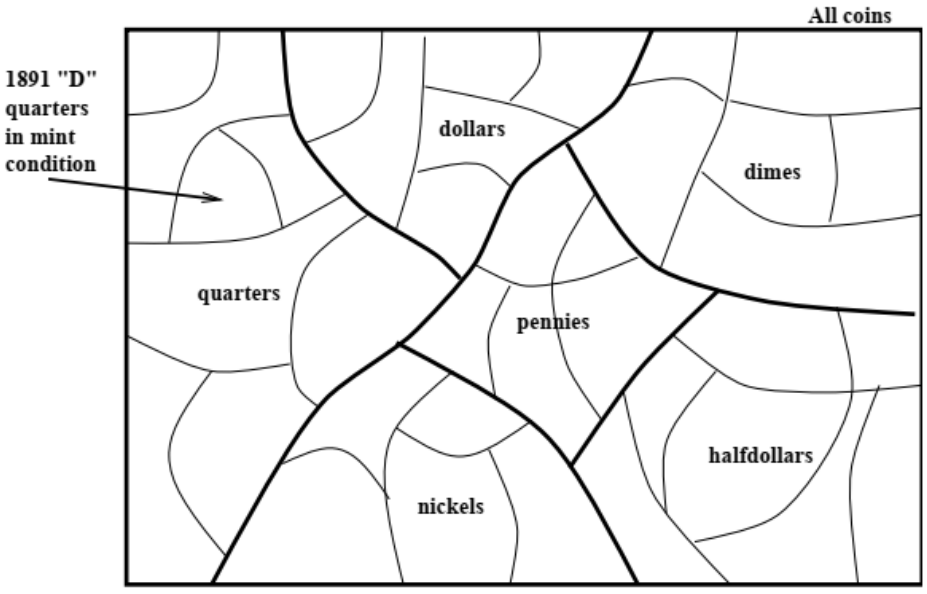

Figure 2.13: A refinement of a partition

Refinements Just exactly what distinctions are of interest may vary. To someone who is collecting rare coins, one quarter is not always going to look as good as the next, but one quarter that was minted in 1891 and has a D on it and is in mint condition looks as good as the next such quarter. Our original equivalence class of all quarters gets subdivided, capturing the finer distinctions that are of interest to a coin collector. The picture of Figure 2.12 gets more boundary lines added to it. See Figure 2.13.

Equivalence Relations Among Programs What differences between program do we normally ignore? It depends on what we are interested in doing with the programs. If all we want to do is put a printout of their source under the short leg of a wobbly table, programs with equally many pages of source code are equivalent.

A very natural notion of equivalence between programs came up in Chapter 1: $IO_a = IO_b$. A further refinement could take running time into account: $IO_a = IO_b$ and $t_a = t_b$ ("t" for time). A further refinement could in addition include memory usage: $IO_a = IO_b$ and $t_a = t_b$ and $s_a = s_b$ ("s" for space). (Both t and s are functions of the input.) The refinements need not stop here. We might insist that the two programs compile to the same object code. If all we want to do with the program is run it, we won't care about further refinements. What else would we want to do with a program but run it? We might want to modify it. Then two programs, one with comments and one without, are not going to be equivalent to us even if they are indistinguishable at run time. So there is a sequence of half a dozen or so successive refinements of notions of program equivalence, each of which is the right one for some circumstances.

and even uncountably infinite sets.

Next we will apply the concept of partitions and equivalence relations to finite-state machines.

A Game

Let PARITY be the set consisting of all binary strings with an even number of 1s. For example, $10100011 \in$ PARITY, $100 \notin$ PARITY. Consider the following game, played by Bob and Alice.

1. Bob writes down a string x without showing it to Alice.
2. Alice asks a question Q about the string x. The question must be such that there are only finitely many different possible answers. E.g., "What is x?" is out, but "What are the last three digits of x?" is allowed.
3. Bob gives a truthful answer $A_Q(x)$ and writes down a string y for Alice to see. (Alice still has not seen x.)
4. If Alice at this point has sufficient information to know whether or not $xy \in$ PARITY, she wins. If not, she loses.

Can Alice be sure to win this game? For example, if her question was *"What is the last bit of x?"* she might lose the game because an answer "1" and a subsequent choice of $y = 1$ would not allow her to know if $xy \in$ PARITY since x might have been 1 or 11. What would be a better question?

Alice asks *"Is the number of 1s in x even?"*. She is sure to win.

Tired of losing Bob suggests they play the same game with a different language: MORE-0S-THAN-1S, the set of bitstrings that contain more 0s than 1s.

Alice asks *"How many more 0s than 1s does x have?"*. Bob points out that the question is against the rules. Alice instead asks *"Does x have more 0s than 1s?"*. Bob answers *"Yes."*. Alice realizes that x could be any one of 0, 00, 000, 0000, ... and concedes defeat.

Theorem 11 *Alice can't win the* MORE-0S-THAN-1S*-game.*

Proof Whatever finite-answer question Q Alice asks, the answer will be the same for some different strings among the infinite collection $\{0, 00, 000, 0000, \ldots\}$. Let 0^{i4} and 0^j be two strings such that $0 \leq i < j$ and $A_Q(0^i) = A_Q(0^j)$. The Bob can write down $x = 0^i$ or $x = 0^j$, answer Alice's question, and then write $y = 1^i$. Since $0^i 1^i \notin$ MORE-0S-THAN-1S and $0^j 1^i \in$ MORE-0S-THAN-1S, Alice cannot know whether or not $x1^i \in$ MORE-0S-THAN-1S — she loses.

What does this game have to do with finite-state machines? It provides a complete characterization of the languages that are accepted by finite-state machines ...

[4]Note: 0^i means a string consisting of i 0s.

Theorem 12 *Consider Bob and Alice playing their game with a language L.*[5] *The following two statements are equivalent.*

1. *Alice can be sure to win.*
2. *L is accepted by some finite-state machine.*

In fact, the connection between the game and finite-state machines is even stronger.

Theorem 13 *Consider Bob and Alice playing their game with the language L. The following two statements are equivalent.*

1. *There is a question that lets Alice win and that can always be answered with one of k different answers.*

2. *There is a finite-state machine with k states that accepts L.*

To see that *(2)* implies *(1)* is the easier of the two parts of the proof. Alice draws a picture of a k-state machine for L and asks which state the machine is in after reading x. When Bob writes down y, Alice runs M to see whether or not it ends up in an accepting state.

The key to proving that *(1)* implies *(2)* is to understand why some questions that Alice might ask will make her win the game while others won't.

The Relation $\equiv_L$ Why was it that in the PARITY game asking whether the last bit of x was a 1 did not ensure a win? Because there were two different choices for x, $x_1 = 1$ and $x_2 = 11$, which draw the same answer, but for which there exists a string y with $(x_1y \in \text{PARITY}) \not\Leftrightarrow (x_2y \in \text{PARITY})$. In this sense the two strings $x_1 = 1$ and $x_2 = 11$ were "not equivalent" and Alice's question made her lose because it failed to distinguish between two nonequivalent strings. Such nonequivalence of two strings is usually written as $x_1 \not\equiv_{\text{PARITY}} x_2$.

Conversely, two strings x_1 and x_2 are equivalent, $x_1 \equiv_{\text{PARITY}} x_2$, if for all strings y, $(x_1y \in \text{PARITY}) \Leftrightarrow (x_2y \in \text{PARITY})$.

Definition 1 *Let L be any language. Two strings x and x' are* equivalent with respect to L, $x \equiv_L x'$, *if for all strings y, $(xy \in L) \Leftrightarrow (x'y \in L)$.*

The Relation $\equiv_Q$ Consider the question Q that made Alice win the DIVISIBLE-BY-8-game: *"What are the last three digits of x, or, if x has fewer than three digits, what is x?"*.

Definition 2 *Let Q be any question about strings. Two strings x and x' are* equivalent with respect to Q, $x \equiv_Q x'$, *if they draw the same answer, i.e., $A_Q(x) = A_Q(x')$.*

The relation $\equiv_Q$ "partitions" all strings into disjoint subset. This partition "induced by $\equiv_Q$" is illustrated in Figure 2.14.

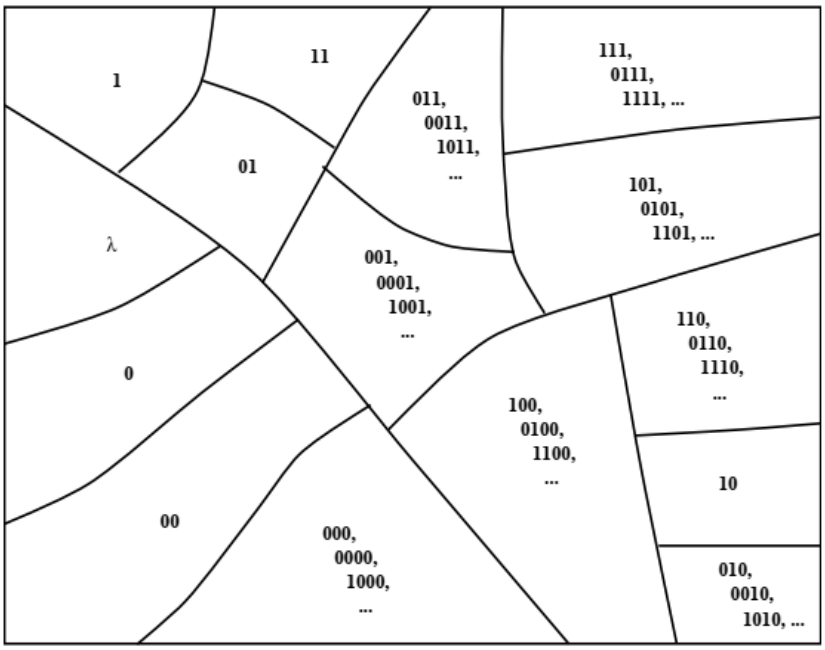

Figure 2.14: The partition induced by Alice's question

The subsets of the partition induced by $\equiv_Q$ then become the state of a finite-state machine. This is illustrated in Figure 2.15. The start state is the state containing the empty string. States containing strings in the language are accepting states. There is a transition labeled 1 from the state that contains 0 to the state that contains 01. Generally, there is a transition labeled c from the state containing s to the state containing sc.

Could it happen that this construction results in more than one transition with the same label out of one state? In this example, we got lucky — Alice asked a very nice question —, and this won't happen. In general it might, though. As an example, Alice might have asked the question, *"If x is empty, say so; if x is nonempty but contains only* 0*s, say so; otherwise, what are the last three digits of x, or, if x has fewer than three digits, what is x?"*. Kind of messy, but better than the original question in the sense that it takes fewer different answers. The problem is that the two strings 0 and 00 draw the same answer, but 01 and 001 don't. Figure 2.16 shows where this causes a problem in the construction of a finite-state machine. The resolution is easy. We arbitrarily choose one of the conflicting transitions and throw away the other(s). Why is this safe? Because $\equiv_Q$ is a "refinement" of $\equiv_L$ and it is safe to ignore distinctions between strings that are equivalent under $\equiv_L$.

The best question

It seems that the finite-state machine in Figure 2.15 is not minimal. We could run our minimization algorithm on it, but let's instead try to work on "mini-

[5]Note: A "language" in this theory means a set of strings.

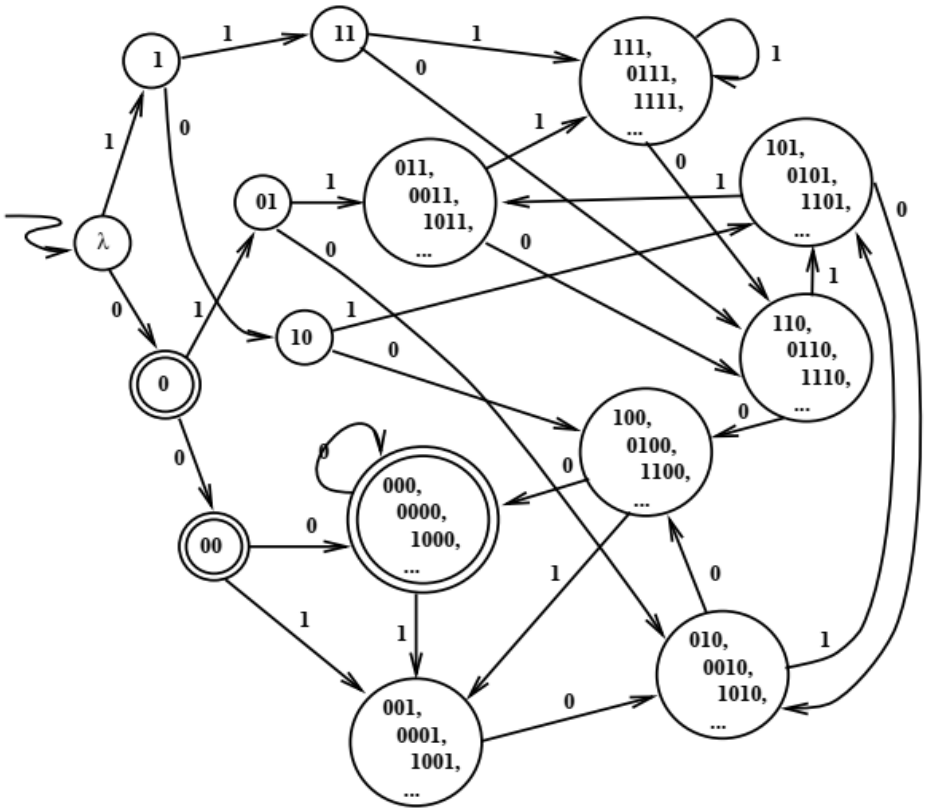

Figure 2.15: The finite-state machine constructed from Alice's question

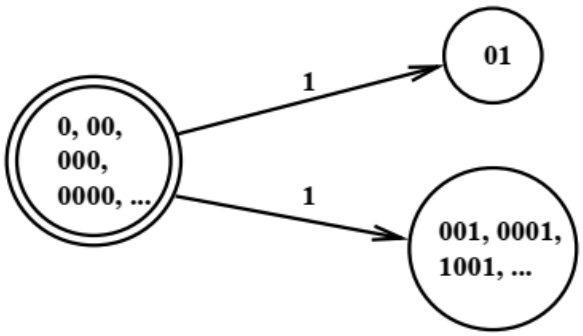

Figure 2.16: A conflict in the construction of a finite-state machine

mizing" the question Q from which the machine was constructed. Minimizing a question for us means minimizing the number of different answers.

The reason why this construction worked was that $\equiv_Q$ was a refinement of $\equiv_L$. This feature of the question Q we need to keep. But what is the refinement of $\equiv_L$ that has the fewest subsets? $\equiv_L$ itself. So the best question is *"Which subset of the partition that is induced by $\equiv_L$ does x belong to?"*. This is easier to phrase in terms not of the subsets but in terms of "representative" strings from the subsets: *"What's a shortest string y with $x \equiv_L y$?"*.

Let's apply this idea to the DIVISIBLE-BY-8 problem. There are four equivalence classes:

$0, 00, 000, 0000, 1000, \ldots$,

$\lambda, 100, 0100, 1100, \ldots$,

$10, 010, 110, 0010, 0110, 1010, 1110, \ldots$, and

$1, 01, 11, 001, 011, 101, 111, 0001, 0011, 0101, 0111, 1001, 1010, 1101, 1111, \ldots$.

The four answers to our best question are $\lambda, 0, 10$, and 1. The resulting finite-state machine has four states and is minimal.

Let's take stock of the equivalence relations (or partitions) that play a role in finite-state machines and the languages accepted by such machines.

The Relations $\equiv_L$, $\equiv_Q$, $\equiv_M$

As mentioned already, every language, i.e., set of strings, L defines ("induces") an equivalence relation $\equiv_L$ between strings:

$$(x \equiv_L y) \stackrel{\text{def}}{\Longleftrightarrow} (\textit{for all strings } z, (xz \in L) \Leftrightarrow (yz \in L)). \tag{2.2}$$

Every question Q about strings defines an equivalence relation $\equiv_Q$ between strings:

$$(x \equiv_Q y) \stackrel{\text{def}}{\Longleftrightarrow} (A_Q(x) = A_Q(y)). \tag{2.3}$$

And every finite-state machine M defines an equivalence relation $\equiv_M$ between strings:

$$(x \equiv_M y) \stackrel{\text{def}}{\Longleftrightarrow} (\delta_M(s_0, x) = \delta_M(s_0, y)). \tag{2.4}$$

s_0 denotes the start state of M. Note that the number of states of M is the number of subsets into which $\equiv_M$ partitions the set of all strings.

As mentioned earlier, for Alice to win the game, her question Q must distinguish between strings x and y for which there exists a string z with $(xz \in L) \not\Leftrightarrow (yz \in L)$. In other words, she must make sure that

$$(x \not\equiv_L y) \Rightarrow (x \not\equiv_Q y) \tag{2.5}$$

i.e., $\equiv_Q$ must be a refinement of $\equiv_L$. If we regard binary relations as sets of pairs, then (2.5) can be written as

$$((x, y) \notin \equiv_L) \Rightarrow ((x, y) \notin \equiv_Q) \tag{2.6}$$

or, for the notationally daring,

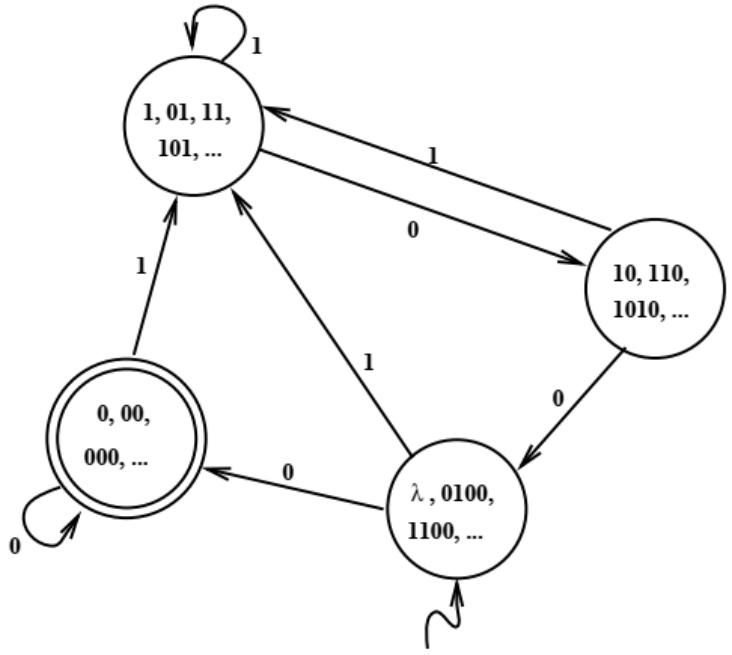

Figure 2.17: Illustrating the uniqueness of a minimal machine

$$\equiv_Q \quad \subseteq \quad \equiv_L \tag{2.7}$$

Instead of talking about Alice and her question, we can talk about a finite-state machine and its states.[6] For the machine M to accept all strings in L and reject all strings that are not in L — a state of affairs usually written as $L(M) = L$ —, M must "distinguish" between strings x and y for which there exists a strings z with $(xz \in L) \not\Leftrightarrow (yz \in L)$. In other words,

$$(x \not\equiv_L y) \Rightarrow (x \not\equiv_M y) \tag{2.8}$$

i.e., $\equiv_M$ must be a refinement of $\equiv_L$. Again, if we regard binary relations as sets of pairs, then (2.8) can be written as

$$\equiv_M \quad \subseteq \quad \equiv_L \tag{2.9}$$

Every machine M with $L(M) = L$ must satisfy (2.9); a smallest machine M_0 will satisfy

$$\equiv_{M_0} \quad = \quad \equiv_L \tag{2.10}$$

Uniqueness How much freedom do we have in constructing a smallest machine for L, given

[6]The analogy becomes clear if you think of a finite-state machine as asking at every step the question *"What state does the input read so far put me in?"*.

that we have to observe (2.10)? None of any substance. The number of states is given, of course. Sure, we can choose arbitrary names for the states. But (2.10) says that the states have to partition the inputs in exactly one way. Each state must correspond to one set from that partition. Figure 2.17 illustrates this for the language DIVISIBLE-BY-8. This then determines all the transitions, it determines which states are accepting, which are rejecting, and which is the start state. Thus, other than choosing different names for the states and laying out the picture differently on the page, there is no choice. This is expressed in the following theorem.

Theorem 14 *All minimal finite-state machines for the same language L are isomorphic.*

Unlike our earlier construction of a finite-state machine from a (non-minimal) winning question of Alice's, this construction will never result in two transitions with the same label from the same state. If that were to happen, it would mean that there are two strings x and y and a character c with $x \equiv_L y$ and $xc \not\equiv_L yc$. Since $xc \not\equiv_L yc$ there is a string s with $(xcs \in L) \not\Leftrightarrow (ycs \in L)$ and therefore a string z $(= cs)$ with $(xz \in L) \not\Leftrightarrow (yz \in L)$, contradicting $x \equiv_L y$.

Figure 2.17 could have been obtained from Figure 2.14 by erasing those lines that are induced by $\equiv_Q$ but not by $\equiv_L$. This is what our minimization algorithm would have done when run on the machine of Figure 2.15.

Summarizing, we have the following theorem.

Theorem 15 (Myhill-Nerode) *Let L be a language. Then the following two statements are equivalent.*

1. *$\equiv_L$ partitions all strings into finitely many classes.*

2. *L is accepted by a finite-state machine.*

Furthermore, every minimal finite-state machine for L is isomorphic to the machine constructed as follows. The states are the subsets of the partition induced by $\equiv_L$. The start state is the set that contains λ. A state A is accepting if $A \subseteq L$. There is a transition from state A to state B labeled c if and only if there is a string x with $x \in A$ and $xc \in B$.