State Equivalence

Robert M. Keller
April 2014
Background

1. With every DFA $M = (Q, \Sigma, \delta, q_0, F)$ there is an associated language $L(M)$ defined by

   $$L(M) = \{ x \in \Sigma^* \mid \delta(q_0, x) \in F \}$$

2. $L(M)$ is the set of strings that take $M$ from its initial state $q_0$ to some accepting state (element of $F$).
Generalizing

• With any state $q$ of $M$ we can associate a language $L(q)$:

$$L(q) = \{ x \in \Sigma^* \mid \delta(q, x) \in F \}$$

• $L(q)$ is the set of strings that take $M$ from state $q$ to some accepting state (element of $F$).

• Therefore $L(M) = L(q_0)$. 
State Equivalence

- Two states will be called equivalent \( q \equiv q' \) iff their associated languages are the same:

\[
L(q) = L(q').
\]
Another Viewpoint

• Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA.

• Recall the extended $\delta: Q \times \Sigma^* \rightarrow Q$.

• Then $q \equiv q'$ iff

  $\forall x \in \Sigma^* \ [\delta(q, x) \in F \leftrightarrow \delta(q', x) \in F]$
Example of State Equivalence

Below, $a \equiv c$ and $b \equiv d$.

$q \equiv q'$ iff $\forall x \in \Sigma^* \; \eta(q, x) = \eta(q', x)$
Why we might care

- Equivalent but distinct states represent **redundancy**.

- The presentation of the DFA might be
  - more understandable,
  - easier to analyze, or
  - less expensive to implement (e.g. in hardware) **without redundancy**.
Redundancy Removed

- The second DFA is a simplified version of the first.
Machine Equivalence

- There is no need for q and q’ to be in the same machine.

- Two DFAs are equivalent iff their initial states are equivalent:

  \[ L(q_0) = L(q'_0) \]
Machine Equivalence Example

$L(a) = L(e) = (\{0\} \cup \{1\}\{1\}^{*}\{0\})^{*}$
Possible skipping ahead

- At this point, one could skip ahead to slide 46 to learn about non-regular languages, then return later.
State Equivalence deserves its name

- State equivalence is an equivalence relation. It is:
  
  **Reflexive:** $\forall q \ ( q \equiv q )$

  **Symmetric:** $\forall q \forall q' \ ( q \equiv q' \rightarrow q' \equiv q )$

  **Transitive:** $\forall q \forall q' \forall q'' \ ( ( q \equiv q' \land q' \equiv q'' ) \rightarrow q \equiv q'' )$

Why? All are based on the definition of $\equiv$ in terms of language equality:

$L(q) = L(q)$

If $L(q) = L(q')$ then $L(q') = L(q)$.

If $L(q) = L(q')$ and $L(q') = L(q'')$, then $L(q) = L(q'')$. 
Partitions

- A **partition** of a set Q is a set of subsets (called “blocks”) of Q such that:
  - No block is empty.
  - No two blocks overlap.
  - The union of the blocks is all of Q.

- Example:
  - Suppose Q = \{a, b, c, d, e, f\}
  - These are examples of partitions of Q:
    - \{\{a, b\}, \{c, d, e\}, \{f\}\}
    - \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}\}
    - \{\{a, b, c, d, e, f\}\}
Partition vs. Equivalence Relation

- Every partition determines an equivalence relation.

- Two elements are defined to be equivalent iff they are in the same block.

- $q \equiv q' \text{ iff } \exists B \in P \ (q \in B \land q' \in B)$

- Example: $P = \{\{a, b\}, \{c, d, e\}, \{f\}\}$
  The equivalence relation is: $a \equiv b, c \equiv d \equiv e, f$. 
Partition vs. Equivalence Relation

- Every equivalence relation determines a partition.
- The partition determined by $\equiv$ is given by
  $\{\{q' \mid q' \equiv q\} \mid q \in Q\}$.
- Two elements are in the same block iff they are equivalent.
- Example: $a \equiv b \equiv c, d \equiv e \equiv f$ determines

  \[ P = \{\{a, b, c\}, \{d, e, f\}\} \]
Summary

- Equivalence Relations and Partitions are two different ways of viewing the same thing.

- The blocks of a partition corresponding to an equivalence relation are called the equivalence classes of the relation.
Refinement of Partitions

• Consider two partitions $P$ and $P'$ on the same set. We say that $P$ refines $P'$ (and write $P \leq P'$) iff every block of $P$ is wholly contained in a block of $P$.

• Examples:
  • $\{\{a, b\}, \{c, d, e\}, \{f\}\} \leq \{\{a, b, f\}, \{c, d, e\}\}$
  • $\{\{a, b, f\}, \{c, d, e\}\} \leq \{\{a, b, c, d, e, f\}\}$
  • $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}\} \leq \{\{a, b\}, \{c, d, e\}, \{f\}\}$
Refinement vs. Containment

- Let $P$ and $P'$ be two partitions on the same set. Let $\equiv$ and $\equiv'$ be the corresponding equivalence relations.

- $P$ refines $P'$ iff $\equiv \subseteq \equiv'$, where containment means as a set of pairs.

- That is $q \equiv q' \rightarrow q \equiv' q'$
Example

- \{\{a, b\}, \{c, d, e\}, \{f\}\} \subseteq \{\{a, b, f\}, \{c, d, e\}\}

- As set of pairs:
  \{(a, b), (c, d), (d, e), (c, e),
   (b, a), (d, c), (e, d), (e, c),
   (a, a), (b, b), (c, c), (d, d), (e, e), (f, f)\}
  \subseteq
  \{(a, b), (a, f), (b, f), (c, d), (c, e),
   (b, a), (f, a), (f, b), (d, c), (e, c),
   (a, a), (b, b), (c, c), (d, d), (e, e), (f, f)\}
Proper Refinement

- $P < P'$ (P properly refines P’) means $P \subseteq P'$ and $P \neq P'$.

- This is equivalent to proper containment as a set of pairs.

  $\equiv \subseteq \equiv'$ means $\equiv \subseteq \equiv'$ and $\equiv \neq \equiv'$
Approaching State Equivalence by Successive Approximations

• Define $\Sigma_k = \{x \in \Sigma^* \mid |x| \leq k\}$ (all strings of $k$ or fewer letters).
  Note: $\Sigma^* = \bigcup\{\Sigma_k \mid k \geq 0\}$

• Define **$k$-equivalence** as follows:

  \[ q \equiv_k q' \]

  iff $\forall x \in \Sigma_k \eta(q, x) = \eta(q', x)$

• In other words, two states are $k$-equivalent iff every string of length $k$ or less takes both to an accepting state or neither to an accepting state.
Repeat of the earlier definition of $\eta$ for reference

- Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA.
- Recall the extended $\delta: Q \times \Sigma^* \rightarrow Q$.

- Define $\eta: Q \times \Sigma^* \rightarrow \{0, 1\}$ by
  $\forall x \in \Sigma^*$
  $$\eta(q, x) = 1 \text{ if } \delta(q, x) \in F$$
  $$0 \text{ otherwise}$$

- Then $q \equiv q'$ iff $\forall x \in \Sigma^* \ \eta(q, x) = \eta(q', x)$
0-equivalence

- From previous definition, $q \equiv_0 q'$

  iff $\forall x \in \Sigma_0 \ \eta(q, x) = \eta(q', x)$

- But there is only one string of length 0, namely $\varepsilon$, and $\delta(q, \varepsilon) = q$, therefore

  $q \equiv_0 q'$ iff $[q \in F \leftrightarrow q' \in F]$

  i.e. both are accepting or neither is.
Equivalence is $k$-equivalence for all $k$

- $q \equiv q'$ is the same as
  $$\forall k \geq 0 (q \equiv_k q')$$

- Reason: Recall that $q \equiv q'$ says
  $$\forall x \in \Sigma^* \eta(q, x) = \eta(q', x)$$
  whereas $q \equiv_k q'$ says
  $$\forall x \in \Sigma_k \eta(q, x) = \eta(q', x)$$

But $\Sigma^* = \cup \{\Sigma_k | k \geq 0\}$. 
More Facts about $\equiv_k$

- $\forall k \geq 0 [(q \equiv_{k+1} q') \rightarrow (q \equiv_k q')]$

- This is obvious since $\Sigma_k \subseteq \Sigma_{k+1}$.

- Also $(q \equiv_k q') \rightarrow \forall i \leq k (q \equiv_i q')$
  (by downward induction)
Less Obvious Fact about $\equiv_k$

- $\forall k \geq 0$
  
  \[
  \left( q \equiv_0 q' \right) \land \forall \sigma \in \Sigma \left[ \delta(q, \sigma) \equiv_k \delta(q', \sigma) \right]
  \]
  
  $\rightarrow (q \equiv_{k+1} q')$

- In other words, two states are $k+1$ equivalent provided
  
  - they are 0-equivalent and
  
  - for every input letter $\sigma$, their pair of next states are $k$-equivalent.
Corollary (replacing 0 with \( k \))

- \( \forall k \geq 0 \)
  \[ (q \equiv_k q') \land \forall \sigma \in \Sigma \left[ \delta(q, \sigma) \equiv_k \delta(q', \sigma) \right] \]
  \[ \rightarrow (q \equiv_{k+1} q') \]

[from previous slide, because \((q \equiv_k q') \rightarrow (q \equiv_0 q')\)]

- In other words, two states are \( k+1 \) equivalent provided they are \( k \)-equivalent and for every input letter \( \sigma \), their pair of next states are \( k \)-equivalent.
\( \equiv_{k+1} \) in Pictures

\[ q \xrightarrow{\sigma} \delta(q, \sigma) \xrightarrow{x} \]

\[ q' \xrightarrow{\sigma} \delta(q', \sigma) \xrightarrow{x} \]

\( \equiv_0 \)

same outputs (both accepting or neither)

\( \equiv_k \)

same outputs pairwise all along the way
Example: $a \equiv_{2+1} c$
Partitioning Algorithm for Computing $\equiv_{k+1}$ from $\equiv_k$

- Given that is $\equiv_k$ known, consider each pair of states such that $q \equiv_k q'$:
  
  - Check for each $\sigma \in \Sigma$ whether or not $\delta(q, \sigma) \equiv_k \delta(q', \sigma)$.

  - If the answer is always yes, then $q \equiv_{k+1} q'$, otherwise not $q \equiv_{k+1} q'$. 
Example: Computing $\equiv_k$ for various k

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>accepting 3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>accepting 4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

- Use partition $P_k$ to represent $\equiv_k$
  - $P_0 = \{\{3, 4\}, \{1, 2, 5, 6\}\}$
  - $P_1 = \{\{3, 4\}, \{1, 6\}, \{2, 5\}\}$
  - $P_2 = \{\{3, 4\}, \{1, 6\}, \{2, 5\}\}$
Stopping Criterion (proof by algorithm)

• Suppose $\equiv_{k+1} = \equiv_k$ (i.e. $P_{k+1} = P_k$).

• Then $\equiv_k = \equiv$
  (i.e. equivalence $\equiv$ is the same as $k$-equivalence).

• Proof: Consider the partitioning algorithm for computing $\equiv_{k+1}$ from $\equiv_k$. If the result $\equiv_{k+1}$ is the same as $\equiv_k$, then the result of computing $\equiv_{k+2}$ will be the same as $\equiv_{k+1}$. By induction, $\equiv_r$ will be the same as $\equiv_{k+1}$ for every $r \geq k$. Hence $\equiv$ is $\equiv_k$. 
Iterated Partitioning Algorithm for State Equivalence

- Given a DFA:
- Compute $P_0$ by dividing into accepting and non-accepting states.
- Iteratively compute $P_1$, $P_2$, ... using the partitioning algorithm until $P_k = P_{k+1}$.
- The last $P_k$ is the partition corresponding to state equivalence.
Try this:
Compute the maximal partition

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a initial</strong></td>
<td>b</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>g</td>
<td>c</td>
</tr>
<tr>
<td><strong>c accepting</strong></td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>h</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>c</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>e</td>
</tr>
<tr>
<td>h</td>
<td>g</td>
<td>c</td>
</tr>
</tbody>
</table>
Observation about Partitioning

• Consider $P_k$ representing $\equiv_k$.

• Then $\forall k \ P_{k+1} \leq P_k$, i.e. $P_{k+1}$ refines $P_k$.

• Proof: In computing $P_{k+1}$ from $P_k$, two states in a block of $P_k$ can separate in $P_{k+1}$, but they never merge.
Termination Proof for Iterative Partitioning Algorithm

- Consider iteratively computing $P_k$ for increasing $k$ starting with $P_0$.
- **Claim**: For any DFA, an iteration must be reached for which $P_k = P_{k+1}$ (the stopping criterion).
- **Proof**: We already observed $\forall k \ P_{k+1} \leq P_k$. Furthermore, either
  - $P_{k+1} < P_k$ (*strict* refinement) or
  - $P_{k+1} = P_k$.
- If the latter, we can stop. If the former, then $|P_{k+1}| > |P_k|$ because at least one block must split if we don’t have equality.
- A partition cannot split more times than the number of states, so eventually there can be no more strict refinement.
Bound on the number of iterations of the iterated partitioning algorithm

- For an n-state DFA, the number of iterations required is **at most** n-2.
- Consider $P_0$. If it has only one block, then all states are accepting or all states are rejecting. Therefore all states are equivalent.
- So in order for another iteration required, $P_0$ would need at least **two blocks**: $|P_0| \geq 2$.
- Beyond $P_0$, another iteration is required only if $P_0$ splits, meaning that the number of blocks increases by at least one.
- From $|P_0| \geq 2$, we see that $|P_1| \geq 3$, and in general: If $P_k \neq P_{k+1}$ then $|P_k| \geq k+2$ blocks.
Illustration

- Consider a 5-state machine.
- If $P_0 = \{\{1, 2, 3, 4, 5\}\}$ we are done; all states are equivalent.
- Otherwise $P_0 = \{\{1, 2\}, \{3, 4, 5\}\}$ say and $|P_0| \geq 2$.
- Then $P_1 = \{\{1, 2\}, \{3\}, \{4, 5\}\}$ say and $|P_1| \geq 3$.

...$

- P_3 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}, |P_3| \geq 5.
Corollary

- Two states of an $n$-state DFA are equivalent iff they are $(n-2)$-equivalent.
DFA Minimization

- Given that $\equiv$ has been computed for $M$, we can define a minimal equivalent DFA $M'$ as follows:
  - The **states** of $M'$ are the equivalence classes of $\equiv$, i.e. the blocks of the final partition $P_k$.
  - The **initial state** of $M'$ is the block containing the initial state of $M$.
  - The **accepting states** of $M'$ are the blocks that contain accepting states of $M$.
  - The **transition function** $\delta'$ for $M'$ is given by:
    \[
    \forall \sigma \in \Sigma \quad \delta'([q], \sigma) = [\delta(q, \sigma)]
    \]
    where $[q]$ means the equivalence class of $q$. 
Minimization Example

• For the previous machine on the left below, we computed the maximum partition to be \{\{3, 4\}, \{1, 6\}, \{2, 5\}\}.

• The minimal equivalent machine is shown on the right.

<table>
<thead>
<tr>
<th>M</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 initial</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3 accepting</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4 accepting</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M’</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 6} initial</td>
<td>{1, 6}</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>{2, 5}</td>
<td>{2, 5}</td>
<td>{1, 6}</td>
</tr>
<tr>
<td>{3, 4} accepting</td>
<td>{3, 4}</td>
<td>{2, 5}</td>
</tr>
</tbody>
</table>
Try this: Minimize this DFA

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a initial</td>
<td>b</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>g</td>
<td>c</td>
</tr>
<tr>
<td>c accepting</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>h</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>c</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>e</td>
</tr>
<tr>
<td>h</td>
<td>g</td>
<td>c</td>
</tr>
</tbody>
</table>
Infinite-State Acceptors

• Some of the previous ideas make sense even if the state set is infinite.

• For example, equivalence of states, state equivalence classes, and partitions are still meaningful.

• Useful **insights** can be gained from thinking about infinite-state acceptors.
Minimizing an Infinite-State Acceptor

- Suppose we have an infinite-state acceptor and we could somehow compute the equivalence relation.

- Consider the number of equivalence classes, which is called the **rank** of the relation.

- If the rank happens to be **finite**, we could still create an equivalent DFA in principle.

- If the rank is **infinite**, there is no equivalent DFA (remember **F** stands for “finite”), because we cannot merge non-equivalent states together.
Example

• Suppose the infinite state set is \{0, 1, 2, 3, ...\}.

• If the equivalence partition is \{\{0, 2, 4, ...\}, \{1, 3, 5, ...\}\} then we have finite rank and can create an equivalent DFA.

• If the equivalence partition is \{\{0\}, \{1, 2\}, \{3, 4, 5\}, \{6, 7, 8, 9\}, ...\} there is no equivalent DFA.
Viewing a Language as an Acceptor

• Here is way to define an infinite-state acceptor \((Q, \Sigma, \delta, q_0, F)\) for any language \(L\) over \(\Sigma\).
  • Let \(Q = \Sigma^*\).
  • Define \(\delta: Q \times \Sigma \rightarrow Q\) by
    \[
    \forall x \in Q \; \forall \sigma \in \Sigma \; \delta(x, \sigma) = x\sigma
    \]
  • Define \(q_0 = \varepsilon\).
  • Define \(F = L\).
  • Call this the Language Machine for \(L\).
Example

• Suppose \( L = \{x \in \{0, 1\}^* \mid x \text{ contains exactly one 1}\} \)

• We can depict the acceptor as an infinite tree.
Equivalence in the Language Machine

• Recall that in a DFA

\[ q \equiv q' \text{ iff } \forall x \in \Sigma^* \ [\delta(q, x) \in F \iff \delta(q', x) \in F] \]

• In the language machine states are strings, so we rephrase using \( x \) and \( y \) instead of \( q \) and \( q' \):

\[ \forall x, y \in \Sigma^* \quad x \equiv y \text{ iff } [\forall z \in \Sigma^* \ xz \in F \iff yz \in F] \]

This relation \( \equiv \) on \( \Sigma^* \) is called the **Myhill-Nerode Relation**.
Myhill-Nerode Theorem

- L is a finite-state language iff $\equiv$ has a finite set of equivalence classes

(called “finite rank”).

http://en.wikipedia.org/wiki/Myhill-Nerode_theorem
Showing a Language is Not Finite-State

- There are two common ways to show that a language is not finite-state:
  
  - **Pumping Lemma**: which provides a necessary condition to finite-state, so show this condition is violated.
  
  - **Myhill-Nerode Theorem**: which provides a necessary and sufficient condition.
Pumping Lemma

• Please read about the Pumping Lemma in section 1.4 of Sipser.

• It is only useful for showing a language is not regular, not that it is regular.

• I often find the Myhill-Nerode theorem easier for reasoning, and it can be used to show a language is regular or non-regular.
Using the Myhill-Nerode Theorem to Show a Language is **Not** Finite-State

- If \( L \) is finite-state, the relation \( x \equiv y \) iff \( \forall z \in \Sigma^* \ xz \in L \iff yz \in L \) must have finite rank.

- To show that the rank is not finite, we only need to establish that there is an infinite set of strings, no two of which are equivalent.
Non-equivalent Strings

- Equivalence is $x \equiv y$ iff
  $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$

- Therefore **Non-Equivalence** is
  $\neg \forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$

which is, by DeMorgan’s Law

$\exists z \in \Sigma^* \neg (xz \in L \leftrightarrow yz \in L)$

which is equivalent to

$\exists z \in \Sigma^* [(xz \in L \land yz \notin L) \lor (yz \in L \land xz \notin L)]$
Distinguishing Strings

- Previously, we showed that two strings \( x, y \in \Sigma^* \) are not equivalent w.r.t. \( L \) provided there is a \( z \in \Sigma^* \) such that

\[
(xz \in L \land yz \notin L) \lor (yz \in L \land xz \notin L)
\]

- \( z \) is said to distinguish \( x \) from \( y \), and be a **distinguishing string** for the pair \( x, y \).

- \( x \) and \( y \) are called **distinguishable**.
Distinguishing Strings

- If two strings $x$, $y$ have a distinguishing string, then $x$ and $y$ **must take the initial state of any acceptor for the language to two different states**.

- If they did not, then applying the distinguishing string to the resulting state would show a contradiction.

- This is true for finite- and infinite-state acceptors of the language.
Example

Consider the language \( \{0^n1^n \mid n \geq 0\} = \{\varepsilon, 01, 0011, 000111, \ldots\} \).
We claim this language is not finite-state.

We can identify an infinite set of strings, such that any pair in the set can be distinguished.
\{0^n1^n \mid n \geq 0\} \text{ Example continued}

Consider two strings $0^m$ and $0^n$, where $m < n$. These strings are distinguishable by $1^m$, as

\[0^m1^m \in L \land 0^n1^m \notin L\]

There is an infinite set of such strings $0^m$ for each natural number $m$, each distinguishable from the others.

Therefore this language is not finite-state.
Example

Consider the language \( L = \{(01)^n \mid n \geq 0\} \)
= \( \{\varepsilon, 01, 0101, 010101, \ldots\} \).
We claim this language is finite-state.

Consider three sets:

\( L, L\{0\}, \) and Other (everything else)

Claim: any pair of strings from one of these two sets are indistinguishable.
\{(01)^n \mid n \geq 0\} \text{ Example Continued}

- \(x, y \in L\): Consider any \(z\).
  - If \(z = 0w\) for some \(w\), then \(x_0\) and \(y_0\) are both in \(L\{0\}\) and \(w\) does not distinguish them.
  - If \(z = 1w\), then \(x_1\) and \(y_1\) are both in \(\text{Other}\), and \(w\) does not distinguish them.
  - Either way, \(z\) does not distinguish \(x, y\).

- \(x, y \in L\{0\}\): Consider any \(z\).
  - If \(z = 1w\) for some \(w\), then \(x_1\) and \(y_1\) are both in \(L\) and \(w\) does not distinguish them.
  - If \(z = 0w\), then \(x_0\) and \(y_0\) are both in \(\text{Other}\), and \(w\) does not distinguish them.
  - Either way, \(z\) does not distinguish \(x, y\).

- \(x, y \in \text{Other}\): Any \(z\) takes them both to \(\text{Other}\).
Diagram

- This diagram summarizes the preceding argument. (Does it look familiar?)
The Minimal Acceptor for Any Language

- Analogous to constructing the minimal equivalent DFA, the minimal acceptor for any language \( L \) consists of using the Myhill-Nerode equivalence classes as states.
- For any string \( x \), let \([x]\) denote its equivalence class.
- The transitions of the minimal acceptor are defined by:
  \[
  \forall x \in \Sigma^* \quad \forall \sigma \in \Sigma \quad \delta([x], \sigma) = [x\sigma]
  \]
- The accepting states are defined by
  \[
  F = \{[x] \mid x \in L\}
  \]
- The initial state is defined as \([\varepsilon]\).
Example: Minimal acceptor for \( \{0^n1^n \mid n \geq 0\} \)

- No two strings \( 0^n \) for different \( n \) are equivalent.
- Is there any other string that is equivalent to \( 0^n \)?
  
  i.e. is there another \( x \) such that 
  \[
  \forall z \ [0^n z \in L \iff xz \in L]
  \]

- Such an \( x \) cannot start with 0, because \( 1^n \) would distinguish it from \( 0^n \).
- Similarly, \( x \) cannot start with 1, because \( \epsilon \) would distinguish it from \( 0^n \).
- Thus there is no such \( x \).
Example: Minimal acceptor for \(\{0^n1^n \mid n \geq 0\}\), continued

- Therefore each string \(0^n\) is in an equivalence class by itself.

- Now consider strings of the form \(0^n1^m\) where \(m < n\). Here the situation is different. For example \(0^51^3 \equiv 0^61^4\).

- Similarly, any two strings \(0^m1^n\) with the same value of \(m-n\) are equivalent.

- Continuing this type of reasoning leads to a minimal-state acceptor diagrammed on the next page.
Congruence

- The relation $\equiv_L$ has the additional property of being a congruence:

  $x \equiv_L y$ implies $\forall z \in \Sigma^* (xz \equiv_L yz)$.

- By induction, a necessary and sufficient condition for $\equiv_L$ to be a congruence is:

  $x \equiv_L y$ implies $\forall \sigma \in \Sigma (x\sigma \equiv_L y\sigma)$. 
Proof that $\equiv_L$ has the congruence property.

- $x \equiv_L y$ means $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.

- We want to show this implies $\forall \sigma \in \Sigma (x\sigma \equiv_L y\sigma)$.

- Suppose $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.
- Let $\sigma \in \Sigma$. We need to show $\forall w \in \Sigma^* ((x\sigma)w \in L \leftrightarrow (y\sigma)w \in L)$.
- Let $w \in \Sigma^*$. Then using $\forall E$ from the supposition, $x(\sigma w) \in L \leftrightarrow y(\sigma w) \in L$.
  Furthermore $x(\sigma w) = (x\sigma)w$ and $y(\sigma w) = (y\sigma)w$.
  The result follows from $\forall I$. 
Congruence Pictured

Applying the same input sequence to congruent states yields congruent states.

Partition of the congruence
Language in terms of Equivalence Classes

- Consider a finite-state language.
- Its Myhill-Nerode equivalence relation must be finite-rank (finite # of classes).
- The language itself must be the union of some of the equivalence classes.

  e.g. $W \cup X$ could be the language
Language in terms of Equivalence Classes

- Is not generally the case that the language is just **one** of the equivalence classes.

- That would be equivalent to requiring a DFA have only one accepting state.