Massachusetts Institute of Technology
6.042J/18.062J, Spring ’16: Mathematics for Computer Science
Prof. Albert R Meyer

Staff Solutions to In-Class Problems Week 6, Wed.

STAFF NOTE: Halting Problem; Ch. 8.2

Problem 1.
String procedures are one-argument procedures that apply to strings over the ASCII alphabet. If application of procedure, $P$, to string $s$ results in a computation that eventually halts, we say that $P$ recognizes $s$. We define $\text{lang}(P)$ to be the set of strings or language recognized by $P$:

$$\text{lang}(P) := \{s \in \text{ASCII}^* \mid P \text{ recognizes } s\}.$$ 

A language is unrecognizable when it is not equal to $\text{lang}(P)$ for any procedure $P$.

A string procedure declaration is a text, $s \in \text{ASCII}^*$, that conforms to the grammatical rules for programs. The declaration defines a procedure $P_s$, which we can think of as the result of compiling $s$ into an executable object. If $s \in \text{ASCII}^*$ is not a grammatically well-formed procedure declaration, we arbitrarily define $P_s$ to be the string procedure that fails to halt when applied to any string. Now every string defines a string procedure, and every string procedure is $P_s$ for some $s \in \text{ASCII}^*$.

An easy diagonal argument in the text showed that $\text{No-halt} := \{s \mid P_s \text{ applied to } s \text{ does not halt}\} = \{s \mid s \not\in \text{lang}(P_s)\}$ is not recognizable.

It may seem pretty weird to apply a procedure to its own declaration. Are there any less weird examples of unrecognizable set? The answer is “many more.” In this problem, we’ll show three more:

- $\text{No-halt-\lambda} := \{s \mid P_s \text{ applied to } \lambda \text{ does not halt}\} = \{s \mid \lambda \not\in \text{lang}(P_s)\}$.
- $\text{Finite-halt} := \{s \mid \text{lang}(P_s) \text{ is finite}\}$.
- $\text{Always-halt} := \{s \mid \text{lang}(P_s) = \text{ASCII}^*\}$.

Let’s begin by showing how we could use a recognizer for $\text{No-halt-\lambda}$ to define a recognizer for $\text{No-halt}$. That is, we will “reduce” the weird problem of recognizing $\text{No-halt}$ to the more understandable problem of recognizing $\text{No-halt-\lambda}$. Since there is no recognizer for $\text{No-halt}$, it follows that there can’t be one for $\text{No-halt-\lambda}$ either.

Here’s how this reduction would work: suppose we want to recognize when a given string $s$ is in $\text{No-halt}$. Revise $s$ to be the declaration of a slightly modified procedure $P_{s'}$ which behaves as follows:

- $P_{s'}$ applied to argument $t \in \text{ASCII}^*$, ignores $t$, and simulates $P_s$ applied to $s$.

So, if $P_s$ applied to $s$ halts, then $P_{s'}$ halts on every string it is applied to, and if $P_s$ applied to $s$ does not halt, then $P_{s'}$ does not halt on any string it is applied to. That is,

$$s \in \text{No-halt} \implies \text{lang}(P_{s'}) = \emptyset$$

$$\implies \lambda \not\in \text{lang}(P_{s'})$$

$$\implies s' \not\in \text{No-halt-\lambda},$$

$$s \not\in \text{No-halt} \implies \text{lang}(P_{s'}) = \text{ASCII}^*$$

$$\implies \lambda \in \text{lang}(P_{s'})$$

$$\implies s' \not\in \text{No-halt-\lambda}.$$
In short,
\[ s \in \text{No-halt} \iff s' \in \text{No-halt-} \lambda. \]

So to recognize when \( s \in \text{No-halt} \) all you need to do is recognize when \( s' \in \text{No-halt-} \lambda \). As already noted above (but we know that remark got by several students, so we’re repeating the explanation), this means that if \( \text{No-halt-} \lambda \) was recognizable, then \( \text{No-halt-} \lambda \) would be as well. Since we know that \( \text{No-halt} \) is unrecognizable, then \( \text{No-halt-} \lambda \) must also be unrecognizable, as claimed.

(a) Conclude that \( \text{Finite-halt} \) is unrecognizable.

Hint: Same \( s' \).

Solution. For \( s' \) as above, we know

\[
\begin{align*}
s \in \text{No-halt} & \implies \text{lang}(P_{s'}) = \emptyset \\
& \implies s' \in \text{Finite-halt}, \\
s \notin \text{No-halt} & \implies \text{lang}(P_{s'}) = \text{ASCII}^* \\
& \implies s' \notin \text{Finite-halt}.
\end{align*}
\]

So to recognize when \( s \in \text{No-halt} \) all you need to do is recognize when \( s' \in \text{Finite-halt} \). ■

Next, let’s see how a reduction of \( \text{No-halt} \) to \( \text{Always-halt} \) would work. Suppose we want to recognize when a given string \( s \) is in \( \text{No-halt} \). Revise \( s \) to be the declaration of a slightly modified procedure \( P_{s''} \) which behaves as follows:

When \( P_{s''} \) is applied to argument \( t \in \text{ASCII}^* \), it simulates \( P_s \) applied to \( s \) for up to \( |t| \) “steps” (executions of individual machine instructions). If \( P_s \) applied to \( s \) has not halted in \( |t| \) steps, then the application of \( P_{s''} \) to \( t \) halts. If \( P_s \) applied to \( s \) has halted within \( |t| \) steps, then the application of \( P_{s''} \) to \( t \) runs forever.

(b) Conclude that \( \text{Always-halt} \) is unrecognizable.

Hint: Explain why

\[ s \in \text{No-halt} \iff s'' \in \text{Always-halt}. \]

Solution.

\[
\begin{align*}
s \notin \text{No-halt} & \implies \text{lang}(P_{s''}) = \{ t \mid |t| \leq \#\text{steps until } P_s \text{ halts on } s \} \\
& \implies \text{lang}(P_{s''}) \text{ is finite} \\
& \implies s'' \notin \text{Always-halt}, \\
s \in \text{No-halt} & \implies \text{lang}(P_{s''}) = \text{ASCII}^* \\
& \iff s'' \in \text{Always-halt}.
\end{align*}
\]

In short,

\[ s \in \text{No-halt} \iff s'' \in \text{Always-halt}. \]

So to recognize when \( s \in \text{No-halt} \) all you need to do is recognize when \( s'' \in \text{Always-halt} \). But since there is no recognizer \( \text{No-halt} \), there can’t one for \( \text{Always-halt} \). ■

(c) Explain why \( \text{Finite-halt} \) is unrecognizable.

Hint: Same \( s'' \).
Solution. We have from the solution to part (b) that

\[ s \notin \text{No-halt} \implies \text{lang}(P_{s''}) = \{ t \mid |t| \leq \text{steps until } P_s \text{ halts on } s \} \]
\[ \implies \text{lang}(P_{s''}) \text{ is finite} \]
\[ \implies s'' \in \text{Finite-halt} \]
\[ \text{iff } s'' \notin \text{Finite-halt}, \]
\[ s \in \text{No-halt} \implies \text{lang}(P_{s''}) = \text{ASCII}^* \]
\[ \implies s'' \in \overline{\text{Finite-halt}}. \]

In short,

\[ s \in \text{No-halt} \text{ iff } s'' \in \overline{\text{Finite-halt}}, \]

which implies that \( \overline{\text{Finite-halt}} \) is unrecognizable.

Note that it’s easy to recognize when \( P_s \) does halt on \( s \): just simulate the application of \( P_s \) to \( s \) until it halts. This shows that \( \overline{\text{No-halt}} \) is recognizable. We’ve just concluded that \( \text{Finite-halt} \) is nastier: neither it nor its complement is recognizable.

Problem 2.
This problem provides a proof of the [Schröder-Bernstein] Theorem:

If \( A \text{ inj } B \) and \( B \text{ inj } A \), then \( A \bij B \).

(1)

Since \( A \text{ inj } B \) and \( B \text{ inj } A \), there are are total injective functions \( f : A \to B \) and \( g : B \to A \).

Assume for simplicity that \( A \) and \( B \) have no elements in common. Let’s picture the elements of \( A \) arranged in a column, and likewise \( B \) arranged in a second column to the right, with left-to-right arrows connecting \( a \) to \( f(a) \) for each \( a \in A \) and likewise right-to-left arrows for \( g \). Since \( f \) and \( g \) are total functions, there is exactly one arrow out of each element. Also, since \( f \) and \( g \) are injections, there is at most one arrow into any element.

So starting at any element, there is a unique and unending path of arrows going forwards (it might repeat). There is also a unique path of arrows going backwards, which might be unending, or might end at an element that has no arrow into it. These paths are completely separate: if two ran into each other, there would be two arrows into the element where they ran together.

This divides all the elements into separate paths of four kinds:

(i) paths that are infinite in both directions,

(ii) paths that are infinite going forwards starting from some element of \( A \).

(iii) paths that are infinite going forwards starting from some element of \( B \).

(iv) paths that are unending but finite.

(a) What do the paths of the last type (iv) look like?

Solution. An even-length cycle of alternating \( f \)- and \( g \)-arrows.

(b) Show that for each type of path, either

(i) the \( f \)-arrows define a bijection between the \( A \) and \( B \) elements on the path, or

(ii) the \( g \)-arrows define a bijection between \( B \) and \( A \) elements on the path, or
(iii) both sets of arrows define bijections.

For which kinds of paths do both sets of arrows define bijections?

**Solution.** For paths that start at a point in \(A\), there will be an \(f\)-arrow out of every point on the path, so the \(f\)-arrows will define a bijection from the \(A\) elements to the \(B\) elements on the path. The \(g\)-arrows don’t define a bijection the other way, because they don’t hit the starting point.

For paths that start at a point in \(B\), the \(g\)-arrows will define a bijection from the \(B\) elements to the \(A\) elements, by the same reasoning.

For the other two types of path—cycles and two-way infinite—every point in \(B\) has exactly one \(f\)-arrow coming in, so these arrows define a bijection from the \(A\) elements to the \(B\) elements. Likewise, the \(g\)-arrows define a bijection from \(B\) to \(A\).

\(\blacksquare\)

(c) Explain how to piece these bijections together to form a bijection between \(A\) and \(B\).

**Solution.** Define a bijection \(h : A \rightarrow B\) as follows:

\[
h(a) := \begin{cases} f(a) & \text{if } a\text{'s path does not start at a point in } B, \\ g^{-1}(a) & \text{otherwise.}\end{cases}
\]

By part (b), \(h\) is a combination of bijections defined by different non-overlapping paths, and since every point is on a unique path used by \(h\), the function \(h\) must be a bijection. \(\blacksquare\)

(d) Justify the assumption that \(A\) and \(B\) are disjoint.

**Solution.** We can always find sets \(A'\) bij \(A\) and \(B'\) bij \(B\) such that \(A'\) and \(B'\) are disjoint. For example, let \(A' = A \times \{0\}\) and \(B' = B \times \{1\}\). Then if we prove (1) for \(A'\) and \(B'\), we could conclude it held for \(A\) and \(B\) because

\[A\text{ bij } A'\text{ bij } B'\text{ bij } B.\] \(\blacksquare\)