Solutions to In-Class Problems Week 3, Fri.

Problem 1.
Let's refer to a programming procedure (written in your favorite programming language —C++, or Java, or Python, ...) as a *string procedure* when it is applicable to data of type `string` and only returns values of type `boolean`. When a string procedure, P, applied to a `string`, s, returns **True**, we'll say that P *recognizes* s. If $\mathcal{R}$ is the set of strings that P recognizes, we'll call P a *recognizer* for $\mathcal{R}$.

(a) Describe how a recognizer would work for the set of strings containing only lower case Roman letter —`a,b,...,z` —such that each letter occurs twice in a row. For example, `aaccaabbzz`, is such a string, but `abb`, `00bb`, `AAbb`, and `a` are not. (Even better, actually write a recognizer procedure in your favorite programming language).

Solution. All the standard programming languages have built-in operations for scanning the characters in a string. So simply write a procedure that checks an input string left to right, verifying that successive pairs of characters in the string are duplicated, lowercase roman characters.

ACTUAL PROGRAM TBA

■

A set of `string`s is called *recognizable* if there is a recognizer procedure for it.

When you actually program a procedure, you have to type the program text into a computer system. This means that every procedure is described by some `string` of typed characters. If a `string`, s, is actually the typed description of some string procedure, let's refer to that procedure as P_s. You can think of P_s as the result of compiling s.[1]

In fact, it will be helpful to associate every string, s, with a procedure, P_s; we can do this by defining P_s to be some fixed string procedure —it doesn't matter which one —whenever s is not the typed description of an actual procedure that can be applied to `string`s. The result of this is that we have now defined a total function, f, mapping every `string`, s, to the set, $f(s)$, of `string`s recognized by P_s. That is we have a total function,

$$f : \texttt{string} \to \mathcal{P}(\texttt{string}). \tag{1}$$

(b) Explain why the actual range of f is the set of all recognizable sets of strings.

[1]The string, s, and the procedure, P_s, have to be distinguished to avoid a type error: you can't apply a string to string. For example, let s be the string that you wrote as your program to answer part (a). Applying s to a string argument, say `oorrmm`, should throw a type exception; what you need to do is apply the procedure P_s to `oorrmm`. This should result in a returned value **True**, since `oorrmm` consists of three pairs of lowercase roman letters

Solution. Since $f(s)$ is the set of strings recognized by P_s, everything in range (f) is a recognizable set. Conversely, every recogizable set is in range (f): if $\mathcal{R}$ is a recognizable set, then by definition, there is a procedure, P, that recognizes R. So if r is the input program from which P was compiled, then $\mathcal{R} = f(r)$. ■

This is exactly the set up we need to apply the reasoning behind Russell's Paradox to define a set that is not in the range of f, that is, a set of strings, $\mathcal{N}$, that is *not* recognizable.

(c) Let

$$\mathcal{N} ::= \{s \in \texttt{string} \mid s \notin f(s)\}.$$

Prove that $\mathcal{N}$ is not recognizable.

Hint: Similar to Russell's paradox or the proof of Theorem **??**.

Solution. By definition of $\mathcal{N}$,

$$s \in \mathcal{N} \quad \text{iff} \quad s \notin f(s). \tag{2}$$

for every `string`, s.

Now assume to the contrary that $\mathcal{N}$ was recognizable by some string procedure. This procedure must have a string, w, that describes it, so we have

$$\begin{aligned} s \in \mathcal{N} \quad &\text{iff} \quad P_w \text{ applied to } s \text{ returns } \textbf{True}, \\ &\text{iff} \quad s \in f(w) \qquad \text{(by def. of } f) \end{aligned} \tag{3}$$

for all `string`'s s.

Combining (2) and (3), we have that for every string, s,

$$s \notin f(s) \quad \text{iff} \quad s \in f(w), \tag{4}$$

for all `string`'s s.

Now letting s be w in (4), we reach the contradiction

$$w \notin f(w) \quad \text{iff} \quad w \in f(w).$$

This contradiction implies that the assumption that $\mathcal{N}$ was recognizable must be false.

■

(d) Discuss what the conclusion of part (c) implies about the possibility of writing "program analyzers" that take programs as inputs and analyze their behavior.

Solution. Let's call a programming procedure "self-unconscious" if it does not return **True** when applied to its own textual definition.

Rephrased informally, the conclusion of part (c) says that it is logically impossible to design a *general* program analyzer, which takes as input the (textual definition) of an arbitrary program, and recognizes when the program is self-unconscious. This implies that it is impossible to write a program which does the more general analysis of how an arbitrary procedure behaves when applied to some given arguments.

BTW, it *is* feasible to write a general procedure that recognizes when an arbitrary input procedure *does* return a value when appiled to the string that describes it —that is, when the procedure is *self-conscious*. The general procedure appllied to input s just simulates P_s applied to s. In other words, this general procedure just acts like a virtual machine simulator or "interpreter" for the programming language of its input programs.

It's also important to recognize that there's no hope of getting around this by switching programming languages. For example, by part (c), no C++ program can analyze arbitrary C++ programs, and no Java program can analyze Java programs, but you might wonder if a language like C++, which allows more intimate manipulation of computer memory than Java, might therefore allow a C++ program to analyze general Java programs. But there is no loophole here: since it's possible to write a Java program that is a simulator/interpreter for C++ programs, if a C++ program could analyze Java programs, so could the Java program that simulated the C++ program, contradicting (c).

It's a different story if we think about the *practical* possibility of writing programming analyzers. The fact that it's logically impossible to write analyzers for completely general programs does not mean that you can't do a very good job analyzing interesting programs that come up in practice. In fact these "interesting" programs are commonly *intended* to be analyzable in order to confirm that they do what they're supposed to do.

So it's not clear how much of a hurdle this theoretical limitation implies in practice. What the theory does provide is some perspective on claims about general analysis methods for programs. The theory tells us that people who make such claims either

- are exaggerating the power (if any) of their methods —say to get a grant or make a sale, or
- are trying to keep things simple by not going into technical limitations they're aware of, or
- perhaps most commonly, are so excited about some useful practical successes of their methods, that they haven't bothered to think about their limitations.

So from now on, if you hear people making claims about completely general program analysis/verification/optimization methods, you'll know they can't be telling the whole story. ■

Problem 2.
The Axiom of Choice can say that if s is a set whose members are nonempty sets that are *pairwise disjoint* —that is no two sets in s have an element in common —then there is a set, c, consisting of exactly one element from each set in s.

In formal logic, we could describe s with the formula,

$$\text{pairwise-disjoint}(s) ::= \quad \forall x \in s.\, x \neq \emptyset QAND \forall x, y \in s.(x \neq y) \text{ IMPLIES } (x \cap y = \emptyset).$$

Similarly we could describe c with the formula

$$\text{choice-set}(c, s) ::= \quad \forall x \in s.\, \exists! z.\; z \in c \cap x.$$

Here "$\exists! z.$" is fairly standard notation for "there exists a *unique* z.

Now we can give the formal definition:

Definition (Axiom of Choice)**.**

$$\forall s.\, \text{pairwise-disjoint}(s) \text{ IMPLIES } \exists c.\, \text{choice-set}(c, s).$$

The only issue here is that Set Theory is technically supposed to be expressed in terms of *pure* formulas in the language of sets, which means formula that uses only the membership relation, $\in$, propositional connectives, and the two quantifies $\forall$ and $\exists$. Verify that the Axiom of Choice can be expressed as a pure formula, by explaining how to replace all impure subformulas above with equivalent pure formulas.

For example, the formula $x = y$ could be replaced with the pure formula $\forall z.\, z \in x$ IFF $z \in y$.

Solution. Here is how the impure subformulas used in the above definition of the Axiom of Choice can be translated into pure formulas:

$$x \neq \emptyset translatesinto \exists y /\, y \in x.$$

$$[x \cap y = \emptyset] translatesinto \, \text{NOT}(\exists z.\, z \in x \text{ AND } z \in y).$$

$$[z \in x \cap y] translatesinto z \in x \text{ AND } z \in y.$$

$$\exists ! z.\, P(z) translatesinto \exists z.\, P(z) \text{ AND } \forall w.\, P(w) \text{ IMPLIES } w = z.$$

This last formula is not pure because it uses $=$, but this is ok since we know it can be replaced by a pure formula.

■

Problem 3.
There are lots of different sizes of infinite sets. For example, starting with the infinite set, $\mathbb{N}$, of nonnegative integers, we can build the infinite sequence of sets

$$\mathbb{N},\ \mathcal{P}(\mathbb{N}),\ \mathcal{P}(\mathcal{P}(\mathbb{N})),\ \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))),\ \ldots.$$

By Theorem **??** from the Notes, each of these sets is *strictly bigger*[2] than all the preceding ones. But that's not all: if we let U be the union of the sequence of sets above, then U is strictly bigger than every set in the sequence! Prove this:

Lemma. *Let $\mathcal{P}^n(\mathbb{N})$ be the nth set in the sequence, and*

$$U ::= \bigcup_{n=0}^{\infty} \mathcal{P}^n(\mathbb{N}).$$

Then

[2]Reminder: set A is *strictly bigger* than set B just means that A surj B, but NOT(B surj A).

1. U surj $\mathcal{P}^n(\mathbb{N})$ *for every* $n \in \mathbb{N}$, *but*
2. *there is no* $n \in \mathbb{N}$ *for which* $\mathcal{P}^n(\mathbb{N})$ surj U.

Now of course, we could take $U, \mathcal{P}(U), \mathcal{P}(\mathcal{P}(U)), \dots$ and can keep on indefinitely building still bigger infinities.

Solution. Everything follows from a trivial observation: if $A \supseteq B$, then A surj B. (Why is this trivial?)

So since $U \supseteq \mathcal{P}^n(\mathbb{N})$, we have U surj $\mathcal{P}^n(\mathbb{N})$, which proves 1.

To prove 2, assume to the contrary that $\mathcal{P}^m(\mathbb{N})$ surj U. Now we know from 1 that U surj $\mathcal{P}^{m+1}(\mathbb{N})$. But this implies that

$$\mathcal{P}^m(\mathbb{N}) \text{ surj } \mathcal{P}^{m+1}(\mathbb{N}) = \mathcal{P}(\mathcal{P}^m(\mathbb{N})),$$

contradicting the fact that, by Theorem **??**, a power set of $\mathcal{P}^m(\mathbb{N}))$ is "strictly bigger" than $\mathcal{P}^m(\mathbb{N}))$.

■