1 Fixed-point combinators

1. The most popular encoding of recursion in λ-calculus is using the Y combinator. Another approach is using the simple U combinator (which is not a fixed-point combinator):

   \[ U = \text{df}\; \lambda x. x\; x \]

   (We use =df to denote a definition, as opposed to semantic equivalence.) Using the U combinator, we can define the factorial function as follows:

   \[ \text{fact} = \text{df}\; U(\lambda f. \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \ast ((f\; f)(n - 1))) \]

   (For clarity, we extend pure lambda calculus with conditionals and arithmetic.)

   Prove that \( \text{fact} \) satisfies the following λ-calculus equation:

   \[ \text{fact} = (\lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \ast (\text{fact}(n - 1))) \]

2. Consider the following theorem for characterizing fixed-point combinators themselves as fixed points:

   Let \( G = \lambda y.\lambda f.f\; (y\; f) \). Then \( M \) is a fixed-point combinator if and only if \( M = GM \).

   \[ (1) \]

   (Note: Recall that the following λ-calculus transformation is called the \( \eta \)-reduction rule:

   \[ (\lambda x. M\; x) \rightarrow^\eta M, \]

   where \( x \) does not occur as one of the free variables of \( M \). You are allowed to use \( \eta \)-reduction in this question.)

   subpart (i)
Use Theorem 1 to show that $Y = \text{df} \lambda f.((\lambda x. f(xx))(\lambda x. f(xx)))$ is a fixed-point combinator.

**subpart (ii)**

Use Theorem 1 to show that the U combinator $(\lambda x. x x)$ is a not a fixed-point combinator.

**subpart (iii)**

Prove Theorem 1. (Note that the theorem involves an “if and only if”; consequently, your proof should have two parts.)

## 2 Hoare Logic

In this question, all variables hold real-world integers—i.e., don’t worry about machine arithmetic and overflows.

1. Using Hoare logic, give a proof of the following Hoare triple:

   $\{x \geq 0 \land y > 0\}$

   ```
   r = x;
   q = 0;
   while (r >= y){
       r = r - y;
       q = q + 1;
   }
   \{x = y * q + r \land 0 \leq r < y\}
   ```

   You need only provide an annotation of the form $\{P\}$ for every location in the program; you do not need to show a derivation tree. Accompany your answer with an English description.

2. Using Hoare logic, give a proof that the following sequence of statements swaps the values of $x$ and $y$.

   ```
   x = x + y;
   y = x - y;
   x = x - y;
   ```

   You may use auxiliary variables to denote the initial/final values of $x$ and $y$. Again, you need only supply annotations of the form $\{P\}$ for every location along the program. Accompany your answer with an English description.
3 Denotational Semantics

Consider the simple imperative language defined below:

\[ \text{program} \rightarrow \text{cmd} \]
\[ \text{cmd} \rightarrow \text{Id} := \text{exp} \mid \text{if cond then cmd else cmd fi} \mid \text{cmd};\text{cmd} \]

That is, a program is a command, and a command is an assignment, an if-then-else, or a command followed by another command. (The definitions of expressions and conditions are not needed for this question; you may assume that neither produces side effects.)

The usual way to define the meaning of programs in the language defined above is as a function from an initial store to a final store (where a store is a map from identifiers to values). A denotational semantics that defines program meanings in this way is given below. (Only the meaning (valuation) functions \( P \) and \( C \), for programs and commands, respectively, are given; the meaning functions \( B \) for boolean expressions, and \( E \) for arithmetic expressions are the usual ones.) For clarity, we use \([\_\_]\) instead of the standard \( \boxed{\_\_} \).

\[
\begin{align*}
P[C] &= \lambda \sigma. \ C[C] \ \sigma \\
C[\text{Id} := \text{exp}] &= \lambda \sigma. \ \sigma[\text{Id} \rightarrow \mathbf{E}[\text{exp}]] \ \sigma \\
C[\text{if cond then } C_1 \text{ else } C_2 \text{ fi}] &= \lambda \sigma. \ \text{if } \ \mathbf{B}[\text{cond}] \ \sigma \ \text{then } \ C[C_1] \ \sigma \ \text{else } \ C[C_2] \ \sigma \\
C[C_1;C_2] &= \lambda \sigma. \ C[C_2] (C[C_1] \ \sigma)
\end{align*}
\]

(As is standard, the notation \( \sigma[\text{Id} \rightarrow y] \) is the store \( \sigma \) with the \( \text{Id} \text{ x} \) mapped to \( y \).)

1. Consider extending the language defined above by adding read and write statements as shown below:

\[ \text{cmd} \rightarrow \text{read} (\text{Id}) \mid \text{write} (\text{Id}) \]

You are to write a denotational semantics for this extended language so that the meaning of a program is a function of the form:

\( (\text{initial-store} \times \text{input-int-list}) \rightarrow \text{output-int-list} \)

The idea is that the program is given an initial store and a list of integers (the input), and produces another list of integers (the output). Executing a command of the form \( \text{read}(x) \) removes the left-most value from the input list and assigns the value to \( x \). Executing a command of the form \( \text{write}(x) \) concatenates the current value of \( x \) to the right end of the output list. Assume that the input list of integers is sufficiently long (i.e., do not worry about handling end-of-file errors). You may use the standard list-manipulation functions \( \text{head}, \ \text{tail}, \ \text{and append} \) (function \( \text{append} \) takes two parameters: an integer \( j \) and a list \( L \), and attaches \( j \) to the right end of \( L \)). Be sure to provide or at least explain any other auxiliary functions that you use in your definitions.
2. Write an alternative denotational semantics for this language so that the meaning of a program is a function of the form:

\[(\text{initial-store} \times \text{initial-map}) \rightarrow (\text{final-store} \times \text{final-map})\]

where initial-map and final-map are maps from program points to values. The intention is that initial-map will always be the map that maps all points to the value \(\bot\) (bottom); final-map is to map a point \(p\) in the program to the value produced at that point when the program is executed on the given initial store. A program point is either an assignment statement or a predicate (the condition of an if-then-else). By the value produced at a point, we mean: for an assignment statement, the value assigned to the left-hand-side variable, and for a predicate, the boolean value to which the predicate evaluates. An example is given below (all program points have integer labels).

1: \(x := 0;\)
if 2: \(z > 0\) then 3: \(x := x + 1\) else 4: \(y := 10\) fi;
if 5: \(x = z\) then 6: \(z := z + 1\) else 7: \(z := y\) fi

Initial store: \(x \rightarrow 0, y \rightarrow 0, z \rightarrow 0\)
Final store: \(x \rightarrow 0, y \rightarrow 10, z \rightarrow 1\)
Final map: \(1 \rightarrow 0, 2 \rightarrow false, 3 \rightarrow \bot, 4 \rightarrow 10, 5 \rightarrow true, 6 \rightarrow 1, 7 \rightarrow \bot\)

Be sure to provide or at least explain any auxiliary functions that you use in your definitions.