Exercise 1: Hoare-Logic for Local Variables  Recall the IMP extended with the command

\[ \text{let } x = e \text{ in } c \]

for introducing local variables. (In HW1 you defined the formal operational semantics for this command.) Extend the Hoare-logic rules for IMP with a rule for the above command.

Exercise 2: Smallstep Semantics for Functions  Consider a language IMP-Fun corresponding to IMP extended with functions as follows. First, a function definition is:

\[ f(x,\ldots)c; \text{return } e \]

where \( f \) is the name of the function, \( x,\ldots \) are the formal parameters of the function, \( c \) is the function body and \( e \) the return expression. Assume that there are no “global” variables; that is, all variables that occur in the function body are “local” variables of the function. We extend the language of expressions with a function call command:

\[ x = f(e,\ldots) \]

where \( f \) is the function to be called, and \( e,\ldots \) the actual arguments passed into the call (and the return value is assigned to \( x \)). Finally, an IMP-Fun program \( \Psi \) is a set of function definitions \( f_0,\ldots,f_n \), where \( f_0 \) is a special “main” function that takes zero arguments. Define a small step semantics for IMP-Fun.

Exercise 3: Termination of Simply Typed \( \lambda \)-calculus Programs  Show that the execution always terminates in the call-by-value simply-typed \( \lambda \)-calculus. You need to consider only integer constants, addition, abstraction and application. If you use induction, state precisely on what do you induct.

Hint: You will run into difficulties with the evaluation of an application, because even though you will get by induction that \( e_1 \) and \( e_2 \) both terminate, you cannot show that the application terminates. The solution is to strengthen the induction hypothesis to say not only that the evaluation terminates, but that it terminates with a value that has certain properties (namely, those that allow you to prove the application case).

Exercise 4: Subtyping  Consider the simply-typed \( \lambda \)-calculus with subtyping with the usual rules. Let \( \Gamma \vdash e : \tau \) be the notation for the typing judgment in this type system. Consider that the abstraction is written \( \lambda x.e \), that is, without the type declaration for the formal, with the following typing rule:

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2 \]

Consider now an alternate set of typing rules obtained from the usual one by removing the subsumption rule and adding instead the following restricted form of subsumption:

\[ \Gamma(x) = \tau \quad \tau \leq \tau' \quad \Gamma \vdash_0 x : \tau' \]

Let us write \( \Gamma \vdash_0 e : \tau \) when expression \( e \) can be shown to have type \( \tau \) in this modified type system. Prove the following theorem:

**Theorem 1** For all \( e \) and \( \tau \), if \( \cdot \vdash e : \tau \) then \( \cdot \vdash_0 e : \tau \)

You can ignore the subtyping rules for constants. Make sure you precisely state the induction hypothesis.