1. [10 points] Evaluate the Arith expression (we omit the num and var syntax constructors for brevity):

   \[
   \begin{align*}
   \text{let } & \ x = \text{plus}(3,2) \\
   \text{in } & \ \text{let } \ y = \text{times}(x,3) \\
   \text{in } & \ \text{plus}(y, \text{times}(x,2))
   \end{align*}
   \]

   Use the rules for \(\Rightarrow\), and give derivations of the transitions for the last three steps.

   **Solution:** The expression is given in “concrete syntax” style, but for greater compactness of expression we will use the syntax constructor style.

   \[
   \begin{align*}
   (0) \ & \text{let}(\text{plus}(3,2), x.\text{let}(\text{times}(x,3), y.\text{plus}(y, \text{times}(x,2)))) \Rightarrow \\
   (1) \ & \text{let}(5, x.\text{let}(\text{times}(x,3), y.\text{plus}(y, \text{times}(x,2)))) \Rightarrow \\
   (2) \ & \text{let}(\text{times}(5,3), y.\text{plus}(y, \text{times}(5,2))) \Rightarrow \\
   (3) \ & \text{let}(15, y.\text{plus}(y, \text{times}(5,2))) \Rightarrow \\
   (4) \ & \text{plus}(15, \text{times}(5,2)) \Rightarrow \\
   (5) \ & \text{plus}(15, 10) \Rightarrow \\
   (6) \ & 25
   \end{align*}
   \]

   The derivation of (3) \(\Rightarrow\) (4) is a single rule derivation using rule E3 (the Let instruction). The derivation of (4) \(\Rightarrow\) (5) is a two-step derivation using instruction E2 followed by search rule E5. The derivation of (5) \(\Rightarrow\) (6) is a single rule derivation using E1.

2. [10 points] State the formal induction principle as a logical formula for proving properties of lists of natural numbers, defined by the abstract syntax:

   \[
   \text{list ::= nil | cons(n, list)}
   \]

   **Solution:**

   \[
   P(\text{nil}) \land (\forall l.P(l) \Rightarrow \forall n.P(\text{cons}(n, l))) \Rightarrow \forall l.P(l).
   \]

3. [20 points] Let \(\Rightarrow\) be the small-step transition relation for Arith, Prove that \(\text{plus}(e_1, e_2) \Rightarrow^n \text{plus}(e'_1, e_2)\) if, and only if, \(e_1 \Rightarrow^n e'_1\).

   **Proof:** \([\Rightarrow^n]\) The proof is by induction on \(n\).

   **Case** \(n = 0\): Assume \(\text{plus}(e_1, e_2) \Rightarrow^0 \text{plus}(e'_1, e_2)\). By the definition of \(\Rightarrow\), for \(n = 0\), we have \(\text{plus}(e_1, e_2) = \text{plus}(e'_1, e_2)\), so \(e_1 = e'_1\), and hence \(e_1 \Rightarrow^0 e'_1\).

   **Case** \(n = k + 1\): The by the definition of \(\Rightarrow^n\), there exists an expression \(e\) such that

   \[
   \begin{align*}
   \text{plus}(e_1, e_2) \Rightarrow & \ e \\
   e \Rightarrow^k & \ \text{plus}(e'_1, e_2)
   \end{align*}
   \]

   **Claim:** The first transition is by the left search rule for \(\text{plus}\), \(e = \text{plus}(e', e_2)\) for some \(e'\) such that \(e_1 \Rightarrow e'\).
The transition could not be by the instruction for \( \text{plus} \), since then \( e \) would be a number \( n \), which is a final state, and there is no transition sequence from \( n \) to \( \text{plus}(e_1, e_2) \). If the transition was by the left search rule, then \( e_1 \) would have to be a value \( n \) and \( e = \text{plus}(n, e_2) \), where \( e_2 \mapsto e_2 \). But then we can prove by induction on \( k \) that all the transitions in the sequence for \( e \mapsto^k \text{plus}(e_1, e_2) \) must also use the right search rule for \( \text{plus} \), and the corresponding premises of these transition rules give a transition sequence \( e_2^{k} \mapsto^1 e_2 \), which is impossible, since we can prove that in Arith, a nonempty transition sequence cannot return to its starting point (this is a corollary of the proof of termination of Arith expression evaluation).

Thus by this claim, \( e = \text{plus}(e_1, e_2) \mapsto^k \text{plus}(e_1, e_2) \). The Induction Hypothesis then says that \( e' \mapsto^k e'_1 \). This together with the assumption that \( e_1 \mapsto e' \) yields \( e_1 \mapsto^m e'_1 \) by the definition of \( \mapsto^m \).

\([\Leftarrow]\): We assume \( e_1 \mapsto^m e'_1 \) and must show that \( \text{plus}(e_1, e_2) \mapsto^n \text{plus}(e'_1, e_2) \). Again the proof is by induction on \( n \).

**Case** \( n = 0 \). Then \( e_1 = e'_1 \) so and hence \( \text{plus}(e_1, e_2) \mapsto^0 \text{plus}(e'_1, e_2) \).

**Case** \( n = k+1 \). Then by the definition of \( \mapsto^m \), there exists an \( e \) such that \( e_1 \mapsto e \) and \( e \mapsto^k e'_1 \). The induction hypothesis is:

\[
\text{(IH)} \quad \text{plus}(e'_1, e_2) \mapsto^k \text{plus}(e'_1, e_2) \]

But \( \text{plus}(e_1, e_2) \mapsto \text{plus}(e'_1, e_2) \) by the left search rule for \( \text{plus} \), and this together with the IH gives \( \text{plus}(e_1, e_2) \mapsto^n \text{plus}(e'_1, e_2) \).

4. [10 points] The self-apply function could be expressed in MinML as \( \text{fun} (x : \tau) \text{ is } \text{apply}(x, x) \). Find a type \( \tau \) such that this is well typed according to the typing rules, or show that this is impossible.

**Solution:** Here we are using the simple, nonrecursive version of function expressions. By the typing rule for a \( \text{fun} \) expression, we would have the premise \( [x : \tau] \vdash \text{apply}(x, x) : \tau' \). In order to derive this premise by the apply rule (the only applicable rule), we would have to have the two premises \( [x : \tau] \vdash x : \tau \rightarrow \tau' \) and also \( [x : \tau] \vdash x : \tau \). The second of these certainly holds for any \( \tau \) by the variable rule. The first would only hold if \( \tau = \tau \rightarrow \tau' \). Since the size of type expression on the left is clearly smaller than the type expression on the right, this is clearly impossible.

5. [10 points] The Small Step evaluation rules for MinML define Call-by-Value evaluation. The Call-by-Name (CBN) version of MinML differs from the one discussed in class in one way: in function applications the function arguments are “passed” before they are evaluated, rather than after evaluation. The primitive operator expressions like \( \text{plus}(e_1, e_2) \) still need to have their arguments evaluated before they can be reduced.

(a) [15 points] Give any new or changed rules for small-step evaluation (\( \mapsto \)) for the CBN MinML.

**Solution:** Rule (9.21), the right search rule for \( \text{apply} \) is dropped, because we don’t want to evaluate the argument of an application. Rule (9.20), the left search rule for \( \text{apply} \) is unchanged since application still requires that the function be evaluated. Rule (9.16), the apply instruction, is replaced by

\[
\frac{(v = \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e)}{\text{apply}(v, e_2) \mapsto \{v, e_2/f, x\}e} \quad (9.16')
\]

All the other rules are unchanged.
(b) [10 points]: Do the typing rules for CBN MinML differ from those of the normal CBV MinML? If so, show the altered typing rules.

**Solution:** The typing rules for Call-By-Name are the same as those for Call-By-Value. Typing judgements tell us about the kind of value computed by an expression, and this value and its type will not change when we change the order of evaluation.

In a purely functional language like MinML, the only aspect of a computation that is affected by the order of evaluation is whether a computation will terminate. Because CBN makes it possible to avoid evaluating a function argument that is not used by the function (e.g. `fun f (x : τ₁) : τ₂ is 3`), the evaluation of more expressions will terminate in CBN. But typing judgements don’t say anything about termination of the expression being typed – they just say that if the evaluation of the expression *does* terminate, the resulting value will have the specified type.