Problem 1: Semantics and Typing for Mini-C (70 points)

In class we considered a Mini-C program that computes the factorial of 10, and discussed how it can typed. This program is listed in Figure 1, and a typing derivation showing that it is well-typed is shown in Figure 2. In this exercise we will use this example to understand syntax, semantics, typing, and type-safety in the context of programming languages. For your reference the syntax, semantics, typing rules, and type-safety theorem of Mini-C are shown in the appendices.

Problem 1.1 (Syntax, 10 points).

1. The program in Figure 1 can clearly be written in the form decl Δ begin c end. What should the declarations Δ and the command c be? (This is a practice problem; it will not be evaluated so you don’t have to turn it in. The answer should be obvious from Figure 2.)

2. Find commands c₁ and c₂ such that the program can be written as decl Δ begin c₁; c₂ end. \[3 \text{ points}\]

3. Find a different pair of commands c₁ and c₂ such that the program can be written as decl Δ begin c₁; c₂ end. \[3 \text{ points}\]

4. Clearly, there is more than one way to parse the program of Figure 1 as decl Δ begin c₁; c₂ end. In practice, what additional information might we associate with the grammar (syntax) of Mini-C to ensure that every program parses in a unique way? (Hint: In Karl Crary’s lecture, how did we resolve ambiguity in parsing \(τ₁ \rightarrow τ₂ \rightarrow τ₃\) and in parsing \(e₁ e₂ e₃\)?) \[4 \text{ points}\]

Problem 1.2 (Semantics or program evaluation, 20 points). A Mini-C program has the form decl Δ begin c end. As discussed in class, Δ only specifies the types of the variables and does not participate in computation, whereas c is the actual “content” of the program that computes. Computation is formalized by a relation \(σ; c \rightarrow σ'; c'\), which means that
command $c$ simplifies in store $\sigma$ to command $c'$ and updates the store to $\sigma'$. The computation of a program $c$, starting from store $\sigma$, is therefore a sequence of steps of the following form:

\[(\sigma; c) \rightarrow (\sigma_1; c_1) \rightarrow (\sigma_2; c_2) \rightarrow \ldots\]

Let $c$ be the command from Problem 1.1, part 1. Assume that this command starts computing in store $\sigma = m \mapsto 0, f \mapsto 0, c \mapsto 0$.

1. What are $\sigma_1$ and $c_1$? \[5 \text{ points}\]
2. What are $\sigma_2$ and $c_2$? \[6 \text{ points}\]
3. At some point, the program will reach a state of the following form ($X$ and $Y$ represent values):
   \[
   \begin{align*}
   \sigma_k &= m \mapsto X, f \mapsto Y, c \mapsto 5 \\
   c_k &= \text{while } (c <= m) \text{ do } (f = f * c; \ c = c + 1)
   \end{align*}
   \]
   What are $X$ and $Y$? \[6 \text{ points}\]
4. What is the store $\sigma_n$ and the command $c_n$ when the program terminates? (You don’t have to determine the number $n$) \[3 \text{ points}\]

**Problem 1.3** (Typing, 20 points). For the $\Delta$ and $c$ of Problem 1.1, part 1, Figure 2 lists a derivation which shows that the program is well-typed, i.e. $\Delta \vdash c$.

1. For the $c_1$ determined in Problem 1.2, part 1, find a similar derivation which shows that $\Delta \vdash c_1$. \[10 \text{ points}\]
2. For the $c_n$ determined in Problem 1.2, part 4, find a derivation which shows that $\Delta \vdash c_n$. \[3 \text{ points}\]
3. Suppose $c$ is an arbitrary program such that $\Delta \vdash c$. Assume that $\Delta \vdash \sigma$. Is it possible to obtain a command $c'$ by evaluating $\sigma; c$ such that it is not the case that $\Delta \vdash c'$? Answer yes or no, and justify your answer informally. [7 points]

Problem 1.4 (Type-safety, 20 points).

1. Suppose that we were to replace the command $c = c + 1$ in the program of Figure 1 by $c = \text{true}$. [8 points]

   (a) Is the new program safe? (Recall that a program is safe if it never gets stuck, i.e. either it evaluates for ever or it reduces to \text{noop}.)

   (b) Is the new program well-typed?

2. Suppose that we were to replace the command $f = f \times c$ in the program of Figure 1 by $f = \text{true}$. [8 points]

   (a) Is the new program safe?

   (b) Is the new program well-typed?
3. Recall from class that a programming language is called type-safe if every well-typed program in it evaluates safely. In class we stated that Mini-C is type-safe. Is the converse of type-safety true for Mini-C, i.e., is every safe Mini-C program also well-typed? [4 points]

2 Problem 2: Language-based Security (30 points)

This problem is based on the security typed language discussed in class. You may want to review the lecture slides and Volpano-Smith-Irvine paper. Consider the following program:

\[ x := 0; \text{if} \ w = 1 \text{then} \ x := 0 \text{else} \ y := 0 \]

In answering these questions, remember that the type soundness theorem guarantees that every well-typed program satisfies non-interference. However the type system is conservative, i.e. a program may not be well typed and still satisfy non-interference. Use this knowledge to check your answers to questions 1 and 2 of both parts of this question.

Problem 2.1 (15 points). For each of the type assignments below, answer whether this program satisfies non-interference and justify your answer with an informal argument.

1. \( w : \text{L var}, x : \text{L var}, y : \text{H var}, \)
2. \( w : \text{H var}, x : \text{L var}, y : \text{H var}, \)
3. \( w : \text{H var}, x : \text{H var}, y : \text{L var}, \)
4. \( w : \text{H var}, x : \text{L var}, y : \text{L var}, \)

Problem 2.2 (15 points). For each of the type assignments below, answer whether this program is well-typed. Use the typing rules to justify your answer (do not skip steps).

1. \( w : \text{L var}, x : \text{L var}, y : \text{H var}, \)
2. \( w : \text{H var}, x : \text{L var}, y : \text{H var}, \)
A  Syntax of Mini-C

Types

\[ T ::= \text{int} \mid \text{bool} \]

Variables \( x, y \)

Declarations

\[ \Delta ::= x_1 : T_1; \ldots; x_n : T_n \]

Integers

\[ n ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \]

Expressions

\[ e ::= x \mid \text{true} \mid \text{false} \mid n \mid e_1 + e_2 \mid e_1 \ast e_2 \mid e_1 <= e_2 \mid e_1 == e_2 \]

Commands

\[ c ::= \text{noop} \mid x = e \mid c_1; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

Programs

\[ P ::= \text{decl } \Delta \text{ begin } c \text{ end} \]

Values

\[ v ::= \text{true} \mid \text{false} \mid n \]

Stores

\[ \sigma ::= x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \]

B  Dynamic Semantics of Mini-C

The reduction rules of Mini-C are shown in Figure 3.

C  Static semantics (Typing) of Mini-C

Typing rules for Mini-C are shown in Figure 4. Further, we say that a store \( \sigma \) satisfies \( \Delta \), written \( \Delta \vdash \sigma \) if:

1. \((x : \text{int}) \in \Delta \implies (x \mapsto n) \in \sigma \) for some \( n \)
2. \((x : \text{bool}) \in \Delta \implies (x \mapsto \text{true}) \in \sigma \) or \((x \mapsto \text{false}) \in \sigma \)

As an illustration, the derivation which shows how to type the command while \((c <= m) \text{ do } (f = f \ast c; c = c + 1)\) from the factorial program of Figure 1 is shown in Figure 2. Each rule in the derivation is an instance of a rule in Figure 4.

D  Type-Safety

\textbf{Theorem D.1} (Safety for commands). Suppose the following hold:

1. \( \Delta \vdash c \)
2. \( \Delta \vdash \sigma \)
\[
\begin{align*}
\sigma \triangleright e & \leftrightarrow e' \\
(x \mapsto v) \in \sigma & \quad \sigma \triangleright e_1 \leftrightarrow e'_1 \\
\sigma \triangleright e_2 \leftrightarrow e'_2 & \quad \sigma \triangleright v_1 + e_2 \leftrightarrow v_1 + e'_2 \\
\text{add}(n_1, n_2) = n & \quad \sigma \triangleright n_1 + n_2 \leftrightarrow n \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright v_1 \mapsto v & \quad \sigma \triangleright v_1 \leftrightarrow v_1 \leftrightarrow e'_2 \\
\text{mult}(n_1, n_2) = n & \quad \sigma \triangleright n_1 * n_2 \leftrightarrow n \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright v_1 \mapsto v & \quad \sigma \triangleright v_1 \leftrightarrow v_1 \leftrightarrow e'_2 \\
\text{leq}(n_1, n_2) = b & \quad \sigma \triangleright n_1 \leq n_2 \leftrightarrow b \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright v_1 \mapsto v & \quad \sigma \triangleright v_1 \leftrightarrow v_1 \leftrightarrow e'_2 \\
\text{eq}(n_1, n_2) = b & \quad \sigma \triangleright n_1 \mapsto n_2 \leftrightarrow b \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright e_1 = e_2 \leftrightarrow e'_1 = e'_2 & \quad \sigma \triangleright v_1 = e_2 \leftrightarrow v_1 = e'_2 \\
\sigma \triangleright e_1 \leftrightarrow e'_1 & \quad \sigma \triangleright e_2 \leftrightarrow e'_2 \\
\sigma \triangleright e_1 == e_2 \leftrightarrow e'_1 == e'_2 & \quad \sigma \triangleright v_1 == e_2 \leftrightarrow v_1 == e'_2 \\
\end{align*}
\]

\[
\begin{align*}
\sigma \triangleright e \leftrightarrow^* v & \quad \sigma \triangleright c \rightarrow^* c' \quad \sigma \triangleright c_1 \rightarrow^* c'_1 \\
\sigma ; (x = e) \rightarrow \sigma [x \mapsto v]; \text{noop} & \quad \sigma ; (c_1; c_2) \rightarrow^* c'; (c'_1; c_2) \\
\sigma ; \text{(if } e \text{ then } c_1 \text{ else } c_2) \rightarrow c_1 & \quad \sigma ; (\text{if } e \text{ then } c_1 \text{ else } c_2) \rightarrow^* c_1 \\
\sigma ; \text{while } e \text{ do } c \rightarrow \sigma ; (c; \text{while } e \text{ do } c) & \quad \sigma ; \text{while } e \text{ do } c \rightarrow \sigma ; \text{noop} \\
\end{align*}
\]

\text{Figure 3: Reduction semantics of Mini-C}

3. \(\sigma ; c \rightarrow^* \sigma' ; c'\)

Then either \(c' = \text{noop}\) or there are \(\sigma''\) and \(c''\) such that \(\sigma' ; c' \rightarrow \sigma'' ; c''\).
Types for expressions

\[(x : T) \in \Delta \quad \frac{\Delta \vdash x : T}{\Delta \vdash \text{true} : \text{bool}} \quad \frac{\Delta \vdash \text{false} : \text{bool}}{\Delta \vdash n : \text{int}} \]

\[\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \quad \Delta \vdash e_1 + e_2 : \text{int} \]

\[\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \quad \Delta \vdash e_1 * e_2 : \text{int} \quad \Delta \vdash e_1 \leq e_2 : \text{bool} \]

\[\Delta \vdash e_1 : \text{int} \quad \Delta \vdash e_2 : \text{int} \quad \Delta \vdash e_1 == e_2 : \text{bool} \]

Types for commands

\[\Delta \vdash \text{noop} \quad \frac{(x : T) \in \Delta \quad \Delta \vdash e : T}{\Delta \vdash x = e} \quad \Delta \vdash c_1 \quad \Delta \vdash c_2 \quad \Delta \vdash c_1 ; c_2 \]

\[\Delta \vdash e : \text{bool} \quad \Delta \vdash c_1 \quad \Delta \vdash c_2 \quad \Delta \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \]

\[\Delta \vdash e : \text{bool} \quad \Delta \vdash c \quad \Delta \vdash \text{while } e \text{ do } c \]

Types for programs

\[\Delta \vdash c \quad \vdash \text{decl } \Delta \text{ begin } c \text{ end} \]

Figure 4: Type system of Mini-C