Class 16 - Soundness of Static Type Checking

We have informally stated our intuition that the evaluation of a well-typed expression can never get stuck. We call a static type system with this property sound and a programming language with a sound static type system is called strongly statically typed. In the following, we give a formal definition of soundness and prove that our type system indeed satisfies this property.

Soundness of the Simple Type System

Before we can formally define the soundness property, we have to provide the operational semantics for our statically typed language. To this end, we use the substitution-based small-step semantics defined in Figure 1.

With the definitions of the typing relation and the operational semantics in place, we can define the soundness of our type system as the conjunction of two properties: (1) each well-typed expression is either a value or can take an evaluation step, and (2) each evaluation step preserves well-typedness. Formally:

1. **Progress**: for all \( e \in \text{Expr} \), if \( e \) is closed and well-typed, then either \( e \in \text{Val} \) or \( e \rightarrow e' \) for some \( e' \in \text{Expr} \).

2. **Preservation**: for all \( e, e' \in \text{Expr} \) such that \( e \rightarrow e' \), if \( e \) is closed and well-typed then so is \( e' \).

Together, the progress and preservation properties ensure that the evaluation of a well-typed closed expression will either eventually terminate and produce a value, or go on forever. That is, soundness guarantees that the evaluation will never get stuck in an expression that is not a value and that cannot take another evaluation step. In the following, we prove that our type system indeed satisfies progress and preservation. We provide the key ideas of these proofs but leave out some of the details. You are encouraged to fill in these details as an exercise. Even if you do not complete the proofs, you should still try to understand the high-level arguments used in the proof outlines that we provide below. The math is actually quite simple as we only use induction, case splitting, and equational reasoning.
\[
\begin{align*}
\frac{e_1 \to e'_1}{e_1 \text{ bop } e_2 \to e'_1 \text{ bop } e_2} & \quad \text{ SEARCHBOP}_1 \\
\frac{e_1 \to e'_1}{e_1 ? e_2 : e_3 \to e'_1 ? e_2 : e_3} & \quad \text{ SEARCHIF} \\
\frac{e_1 \to e'_1}{e_1 (e_2) \to e'_1 (e_2)} & \quad \text{ SEARCHCALL}_1 \\
\frac{e_2 \to e'_2}{v_1 \text{ bop } e_2 \to v_1 \text{ bop } e'_2} & \quad \text{ SEARCHBOP}_2 \\
\frac{e_d \to e'_d}{\text{ const } x = e_d; e_b \to \text{ const } x = e'_d; e_b} & \quad \text{ SEARCHCONST} \\
\frac{e_1 \to e'_1}{e_1 (e_2) \to e'_1 (e_2)} & \quad \text{ SEARCHCALL}_2 \\
\frac{n = n_1 \text{ bop } n_2 \quad \text{ bop } \in \{+,*\}}{n_1 \text{ bop } n_2 \to n} & \quad \text{ DoArith} \\
\frac{b = v_1 \text{ bop } v_2 \quad \text{ bop } \in \{===,===\}}{v_1 \text{ bop } v_2 \to b} & \quad \text{ DoEqual} \\
\frac{\text{ false } \& \& e_2 \to v_1}{\text{ DoAndFalse}} \\
\frac{\text{ true } || e_2 \to v_1}{\text{ DoOrTrue}} \\
\frac{\text{ true } ? e_2 : e_3 \to e_2}{\text{ DoIfThen}} \\
\frac{\text{ false } ? e_2 : e_3 \to e_3}{\text{ DoIfElse}} \\
\frac{\text{ const } x = v_d; e_b \to e_b[v_d/x]}{\text{ DoConst}} \\
\frac{v_1 = \text{ function } (x: \tau) t e}{v_1 (v_2) \to e[v_2/x]} & \quad \text{ DoCall} \\
\frac{v_1 = \text{ function } x_1 (x_2: \tau_2) : \tau' e}{v_1 (v_2) \to e[v_1/x_1][v_2/x_2]} & \quad \text{ DoCallRec}
\end{align*}
\]

Figure 1: Small-step operational semantics
To prove progress, we first need an auxiliary lemma that characterizes values based on their types:

**Lemma 1 (Canonical Forms).** Let \( v \) be a closed value. Then the following properties hold:

1. If \( v \) is of type \( \text{bool} \), then \( v \in \text{Bool} \).
2. If \( v \) is of type \( \text{number} \), then \( v \in \text{Num} \).
3. If \( v \) is of type \((x:\tau) \Rightarrow \tau'\), then either 
   
   \[ v = \text{function} (x:\tau) t \ e \ \text{where} \ t = \epsilon \ \text{or} \ t = :\tau', \text{or} \]
   
   \[ v = \text{function} \ x_1 (x_2:\tau) : \tau' e \]

*Proof (Exercise).* The lemma follows easily from the rules of the typing relation that involve values. \(\square\)

**Theorem 2 (Progress).** Let \( e \in \text{Expr} \) be a closed expression. If \( e \) is well-typed, then either \( e \in \text{Val} \) or \( e \rightarrow e' \) for some \( e' \in \text{Expr} \).

*Proof (Exercise).* Since \( e \) is well-typed and closed we have \( \emptyset \vdash e : \tau \) for some type \( \tau \). The proof goes by induction on the derivation of \( \emptyset \vdash e : \tau \). We proceed by cases on the final typing rule used in the derivation.

If the last rule that was used in the derivation of \( \emptyset \vdash e : \tau \) is \text{TypeNum}, \text{TypeBool}, \text{TypeFunction}, \text{TypeFunctionAnn}, or \text{TypeFunctionRec}, then \( e \) must be a value and there is nothing to be proved. The rule \text{TypeVar} cannot be the last rule used in the derivation since it would imply that \( e = x \) for some variable \( x \in \text{Var} \), contradicting the assumption that \( e \) is closed. From the remaining cases, we only show the case for the rule \text{TypeCall}. The other cases are similar and left as an exercise.

Thus, assume that \text{TypeCall} is the last rule that was used in the derivation of \( \emptyset \vdash e : \tau \). From the premises of \text{TypeCall}, it follows that \( e \) must be of the form \( e = e_1 (e_2) \) such that \( \emptyset \vdash e_1 : ((x:\tau') \Rightarrow \tau) \) and \( \emptyset \vdash e_2 : \tau' \) for some \( e_1, e_2, x \), and \( \tau' \). We distinguish three subcases:

**Case 1** \( e_1 \) is not a value: since \( e_1 \) is closed and well-typed, \( e_1 \) can take a step by induction hypothesis. That is, there exists \( e_1' \) such that \( e_1 \rightarrow e_1' \). Then by rule \text{SearchCall}, we can conclude \( e \rightarrow e_1' (e_2) \).

**Case 2** \( e_1 \) is a value but \( e_2 \) is not: similar to the previous case, we can conclude by induction hypothesis and rule \text{SearchCall2} that \( e \rightarrow e_1 (e_2') \) for some \( e_2' \).

**Case 3** \( e_1 \) and \( e_2 \) are both values: it follows from the Canonical Forms Lemma that \( e_1 \) must be of the forms

\[ e_1 = \text{function} (x:\tau') t e' \ \text{or} \ e_1 = \text{function} x_1 (x_2:\tau) : \tau e'. \]

In the first case, we conclude from rule \text{DoCall} that \( e \rightarrow e'[e_2/x] \). In the second case, rule \text{DoCallRec} implies that \( e \rightarrow e'[e_1/x_1][e_2/x_2] \).
To prove the preservation property, we first need to state some technical lemmas. We start with two lemmas that allow us to transform typing derivations in specific cases.

First, the Permutation Lemma states that the order in which we extend the typing environment does not matter for a typing derivation as long as the variables for which we extend the environment are distinct.

Lemma 3 (Permutation). If $\Gamma[x \mapsto \tau_1][y \mapsto \tau_2] \vdash e : \tau$ and $x \neq y$, then $\Gamma[y \mapsto \tau_2][x \mapsto \tau_1] \vdash e : \tau$.

Proof (Exercise). The intuition for why this lemma is correct is that if we extend an environment $\Gamma'$ with another binding for a variable $y$, $\Gamma'' = \Gamma'[y \mapsto \tau_2]$, then the second extension does not interfere with the first extension unless $x$ and $y$ are equal.

Thus, define $\Gamma_1 = \Gamma[x \mapsto \tau_1][y \mapsto \tau_2]$ and $\Gamma_2 = \Gamma[y \mapsto \tau_2][y \mapsto \tau_1]$. All you need to prove is that $\Gamma_1 = \Gamma_2$. To do so, prove that for all variables $z$, if $z \in \text{dom}(\Gamma_1)$ then $z \in \text{dom}(\Gamma_2)$ and $\Gamma_1(z) = \Gamma_2(z)$, and vice versa. The actual proof is easy. You only need to expand the definition of the extension function $\cdot \mapsto \cdot$ and then case split on $z$ (i.e., whether $z = x$, $z = y$, or $z \neq x$ and $z \neq y$).

The second technical lemma that we will need is the Weakening Lemma. It states that if an expression $e$ is well-typed under some environment $\Gamma$, then it is also well-typed under any environment $\Gamma' = \Gamma[x \mapsto \tau]'$, provided that $x$ does not occur free in $e$. The intuition for this lemma is that if $x$ occurs at all in $e$, then any such occurrence must be a bound occurrence. However, the mapping of $x$ to $\tau'$ in the environment will be overwritten at each defining occurrence of $x$ in $e$. These updated environments will then be used to type the using occurrences of $x$. Thus, the mapping of $x$ to $\tau'$ will never actually be used in the typing derivation for $e$.

Lemma 4 (Weakening). If $\Gamma \vdash e : \tau$ and $x \notin \text{fv}(e)$, then $\Gamma[x \mapsto \tau'] \vdash e : \tau$.

Proof (Exercise). The proof goes by induction on the derivation of $\Gamma \vdash e : \tau$ and is quite simple.

The core of the proof of the preservation property is the following Substitution Lemma, which states that well-typedness is preserved under substitution. The proof of this Lemma uses the permutation and weakening lemmas.

Lemma 5 (Preservation of Types under Substitutions). If $\Gamma[x \mapsto \tau_2] \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$, then $\Gamma \vdash e_1[e_2/x] : \tau_1$.

Proof (Exercise). The proof goes by induction on the derivation of $\Gamma[x \mapsto \tau'] \vdash e_1 : \tau$. We proceed by cases on the final typing rule used in this derivation. The most interesting cases are the ones for variables, procedural abstraction,
and \texttt{const} declarations. We consider the rules \texttt{TypeVar} and \texttt{TypeFunction} explicitly and leave the remaining rules as an exercise.

If the last rule in the derivation is \texttt{TypeVar}, then $e_1 = y$ and $\Gamma[x \mapsto \tau_2](y) = \tau_1$. There are two subcases based on whether $y$ is $x$ or another variable. If $y = x$, then $e_1[e_2/x] = y[e_2/x] = e_2$ and $\tau_1 = \tau_2$. Then $\Gamma \vdash e_1[e_2/x] : \tau_1$ directly follows from the assumption $\Gamma \vdash e_2 : \tau_2$. If $y \neq x$, then $e_1[e_2/x] = y$ and $\Gamma(y) = \tau_1$. Thus, $\Gamma \vdash e_1[e_2/x] : \tau_1$ follows immediately from rule \texttt{TypeVar}.

If the last rule in the derivation is \texttt{TypeFunction}, then we know that

- $e_1 = \text{function } (y : \tau) \ e,$
- $\tau_1 = (y : \tau) \Rightarrow \tau',$ and
- $\Gamma[x \mapsto \tau_2][y \mapsto \tau] \vdash e : \tau'.$

Since we are allowed to consistently rename bound variables by fresh variables, we may assume that $y \neq x$ and $y \notin \text{fv}(e_2)$. Using $y \neq x$ and the Permutation Lemma, we can derive $\Gamma[y \mapsto \tau][x \mapsto \tau_2] \vdash e : \tau'$ from $\Gamma[x \mapsto \tau_2][y \mapsto \tau] \vdash e : \tau'.$ Using $y \notin \text{fv}(e_2)$ and the Weakening Lemma, we can conclude $\Gamma[y \mapsto \tau] \vdash e_2 : \tau_2$ from $\Gamma \vdash e_2 : \tau_2$. Now, from the induction hypothesis it follows $\Gamma[y \mapsto \tau] \vdash e[e_2/x] : \tau'.$ Thus, by rule \texttt{TypeFunction} we conclude

$$\Gamma \vdash \text{function } (y : \tau) (e[e_2/x]) : (y : \tau) \Rightarrow \tau'.$$

By the definition of substitution and since $y \neq x$, we have

$$\text{function } (y : \tau) (e[e_2/x]) = (\text{function } (y : \tau) \ e)[e_2/x] = e_1[e_2/x]$$

Hence, we conclude $\Gamma \vdash e_1[e_2/x] : \tau_1$.

\begin{theorem}[Preservation] \label{thm:preservation}
Let $e, e' \in \text{Expr}$ such that $e \rightarrow e'$. If $e$ is closed and well-typed, then so is $e'$.
\end{theorem}

\begin{proof}[Exercise]
Since $e$ is closed and well-typed, we must have $\emptyset \vdash e : \tau$ for some type $\tau$. We prove a slightly stronger property than required, namely that if $e \rightarrow e'$, then $e'$ is closed and $\emptyset \vdash e' : \tau$ (i.e., the exact type $\tau$ of $e$ is preserved under evaluation). The proof goes by induction on the derivation of $\emptyset \vdash e : \tau$ using case splitting on the last rule of the derivation. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., whenever $\emptyset \vdash e_1 : \tau_1$ is proved by a subderivation and $e_1 \rightarrow e'_1$, then $e'_1$ is closed and $\emptyset \vdash e'_1 : \tau_1$). We leave the details of the proof as an exercise. \textbf{Hint}: In each case, the final typing rule in the derivation of $\emptyset \vdash e : \tau$ determines the top-level syntactic structure of $e$. In turn, this restricts the possible final rules that may have been used in the derivation of $e \rightarrow e'$. By further case splitting on these relevant final rules of the small-step SOS, you can apply the induction hypothesis where needed. The interesting cases are the rules \texttt{TypeConst} and the typing rules for calls in combination with the relevant do rules of the small-step SOS for these cases. In all these interesting cases, you need to apply the substitution lemma (Lemma 5) to complete the proof.
\end{proof}