CS 185: Type Preservation, Constraint Solving.

Types. \[\langle \text{type} \rangle \ ::= \langle \text{base} \rangle \mid \langle \text{type} \rangle \to \langle \text{type} \rangle\]

Type assignment rules.
\[
\begin{align*}
\Gamma(x) &= T \\
\Gamma \vdash x : T &\quad \text{t-var} \\
\Gamma \vdash t_1, t_2 : T_1 &\quad \text{t-app} \\
\Gamma, x : T_1 \vdash t : T_2 &\quad \text{t-lam}
\end{align*}
\]

Type Preservation Theorem. If \( \Gamma \vdash t : T \) and \( t \sim_\beta t' \) (full \( \beta \)-reduction), then \( \Gamma \vdash t' : T \).

We'll prove this by rule induction on the proof of the reduction \( t \sim_\beta t' \), which we will take as being built using these rules:
\[
\begin{align*}
(\lambda x.t_1) t_2 &\sim t_1 / x \mapsto t_2 \quad \text{r-\( \beta \)} \\
t_1 t_2 &\sim t_1' t_2 \quad \text{r-app1} \\
t_2 &\sim t_2' \quad \text{r-app2} \\
lam x.t &\sim \lam x.t' \quad \text{r-lam}
\end{align*}
\]

We'll do the \( \text{r-\( \beta \)} \) case last.

**Case: r-app1.**

We have
\[
\begin{align*}
t_1 &\sim t_1' \\
t_1 t_2 &\sim t_1' t_2
\end{align*}
\]

By inversion on the typing derivation, we have:
\[
\begin{align*}
\Gamma \vdash t_1 : T_1 \to T_2 &\quad \text{t-app} \\
\Gamma \vdash t_2 : T_2 &
\end{align*}
\]

We may apply our induction hypothesis to the proof of \( t_1 \sim t_1' \) and the proof in the first premise of this inference, to get:
\[
\Gamma \vdash t_1' : T_2 \to T_1
\]

Putting this together with our proof of \( \Gamma \vdash t_2 : T_2 \) (from the second premise of the typing proof above), we have
\[
\Gamma \vdash t_1' t_2 : T_2 \\
\Gamma \vdash t_1 t_2 : T_2
\]

**Case: r-app2:** similar to the \( \text{r-app1} \) case.

**Case: r-lam.**

We have:
\[
\begin{align*}
t &\sim t'
\end{align*}
\]

By inversion on the typing derivation, we also have:
\[ \Gamma, x : T_1 \vdash t : T_2 \]
\[ \Gamma \vdash \lambda x. t : T_1 \rightarrow T_2 \quad t\text{-lam} \]

Then by our induction hypothesis applied to the derivation in the premise of the reduction inference and the derivation in the premise of the typing inference, we obtain:

\[ \Gamma, x : T_1 \vdash t' : T_2 \]

Now we may apply \( t\text{-lam} \) to that to get:

\[ \Gamma \vdash \lambda x. t' : T_1 \rightarrow T_2 \]

Case: \( r\text{-}\beta \).

We have:

\[ (\lambda x. t_1) t_2 \leadsto t_1/x \mapsto t_2 \quad r\text{-}\beta \]

By inversion on the typing derivation, we also have:

\[ \Gamma, x : T_1 \vdash t_1 : T_1 \]
\[ \Gamma \vdash (\lambda x. t_1) : T_2 \rightarrow T_1 \]
\[ \Gamma \vdash t_2 : T_2 \]
\[ \Gamma \vdash (\lambda x. t_1) t_2 : T_1 \]

Here we must apply the following lemma (whose proof we omit):

Lemma 1 (Substitution) If \( \Gamma, x : T_2 \vdash t_1 : T_1 \) and \( \Gamma \vdash t_2 : T_2 \), then \( \Gamma \vdash t_1/x \mapsto t_2 : T_1 \).

Applying this lemma allows us to complete the proof of the Type Preservation Theorem.

Constraint generation. Last time, we saw that the above typing rules are not algorithmic. Whether one is computing a type or checking a type, one must make a non-deterministic choice of a type in the premise of one rule. One solution is to add annotations to the program that specify this type, thus removing the need for the non-deterministic choice.

Another way to get a typing algorithm without adding any annotations is to modify our type assignment rules so they generate constraints. This idea can be implemented based on an interpretation of the typing judgment as expressing type checking, as well as on an interpretation as type computation. Here, we pursue the latter. The rules now operate on judgments of the form \( \Gamma \vdash t : T > C \), where \( \Gamma \) and \( t \) are inputs, and \( T \) and \( C \) are outputs. \( C \) is a set of constraints which must be satisfied in order for the type assignment to hold. A constraint is an equation between simple types with meta-variables \( \langle\text{meta-var}\rangle \), defined by the following syntax:

\[
\langle\text{type}\rangle ::= \langle\text{base}\rangle \mid \langle\text{meta-var}\rangle \mid \langle\text{type}\rangle \rightarrow \langle\text{type}\rangle
\]

The constraint generation rules are the following (where \( \cdot \) denotes the empty set of constraints, and comma is used for unioning sets of constraints). In the rule for \( \lambda \)-abstractions and the rule for applications, \( X \) is a new meta-variable (not occurring in any other term, type, or context listed).
\[ \Gamma(x) = T \quad \Gamma \vdash t_1 : T_1 > C_1 \quad \Gamma \vdash t_2 : T_2 > C_2 \quad \Gamma, x : X \vdash t : T > C \]

\[ \Gamma \vdash t_1 t_2 : X > C_1, C_2, T_1 = T_2 \rightarrow X \quad \Gamma \vdash \lambda x.t : X \rightarrow T > C \]

So to compute a type for a term, one applies these rules bottom-up (from conclusion to premises). This will generate a set of constraints. If these have a common solution, then the original term is typable. To solve the constraints, unification is used.

The syntactic unification problem is the following: given two expressions \( e_1 \) and \( e_2 \) which may use unification variables \( X \) (in our case from \( \langle \text{meta-var} \rangle \)), find a substitution \( \sigma \) for those variables such that \( e_1/\sigma \) and \( e_2/\sigma \) are exactly the same expression. This substitution is a unifier of the two expressions.

An algorithm is given by the following rules, which are applied top-down (from premises to conclusion) to transform a set of constraints \( C \):

\[
\begin{align*}
\frac{t = t, C}{C} & \quad \text{delete} \\
\frac{X = t, C}{t = X, C} & \quad \text{orient} \\
\frac{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)}{t_1 = s_1, \ldots, t_n = s_n, C} & \quad \text{decompose} \\
\frac{X = t, C}{X = t, C/X \mapsto t} & \quad \text{solve}
\end{align*}
\]

A variable is called solved in \( C \) if it occurs exactly once in \( C \), on the left hand side of an equation. A number of important properties of this algorithm can be shown, including that it computes a most general unifier, which is one that does not solve variables unnecessarily: you cannot get a unifier by dropping the equation associated with a solved variable, unless the right hand side is a variable occurring only once (then the equation is just renaming the left hand side variable).

We observe here just that the algorithm terminates, by reducing a certain measure: (\# unsolved variables, size of constraint set, \# unoriented equations), where the size of the constraint set is the sum of the number of symbols except equality in its members; and an equation is unoriented if \( \text{orient} \) could be applied to it. In the table below, a dash indicates a value that could possibly increase (yet in the lexicographic order, the measure is still decreased).

<table>
<thead>
<tr>
<th>Rule</th>
<th># unsolved variables</th>
<th>size of constraint set</th>
<th># unoriented equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>≥</td>
<td>&gt;</td>
<td>-</td>
</tr>
<tr>
<td>decompose</td>
<td>≥</td>
<td>&gt;</td>
<td>-</td>
</tr>
<tr>
<td>orient</td>
<td>≥</td>
<td>=</td>
<td>&gt;</td>
</tr>
<tr>
<td>solve</td>
<td>&gt;</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>