Solving typing constraints

\{ \alpha = \text{int} \rightarrow \beta, \beta = \beta_1 \times \text{bool}, \beta_1 = \text{int} \} \text{ can be solved by}
\[
\alpha = \text{int} \rightarrow (\text{int} \times \text{bool}) \\
\beta = \text{int} \times \text{bool} \quad \beta_1 = \text{int}
\]

\{ \alpha_1 \rightarrow \alpha_2 = \text{int} \rightarrow \beta, \beta = \text{int} \rightarrow \alpha_1 \} \\
\alpha_1 = \text{int} \quad \beta = \text{int} \rightarrow \text{int} = \alpha_2

So we can use something like Gaussian elimination. The algorithm is called \textit{unification}.

We write \( \sigma \) for a substitution \([ \tau/\alpha ]\) where \( \tau \) is a type (perhaps containing type variables) and \( \alpha \) is a type variable. We write \( [\sigma] \tau \) for the effect of carrying out \( \sigma \). If \( \tau_1 \) and \( \tau_2 \) are type expressions & \( \sigma \) is a substitution on all the type variables - so \( \sigma \) could look like \([ \tau_1/\xi_1, \tau_2/\xi_2, \ldots ]\) - such that
\[
[\sigma] \tau_1 = [\sigma] \tau_2, \text{where the equality sign now means identity}, \text{we say that } \tau_1 \text{ & } \tau_2 \text{ are unifiable & } \sigma \text{ is the unifier.}
\]

How do we solve constraints? We transform a set of constraints using the following rules:
\[
\{ C_1, C_2, \ldots C_n, \text{ int} = \text{int} \} \Rightarrow \{ C_1, \ldots C_n \} \\
\{ C_1, C_2, \ldots C_n, \text{ bool} = \text{bool} \} \Rightarrow \{ C_1, \ldots C_n \} \\
\{ C_1, \ldots C_n, \alpha = \tau \} \Rightarrow \{ [\tau/\alpha] C_1, [\tau/\alpha] C_2, \ldots, [\tau/\alpha] C_n \} \\
\{ C_1, \ldots C_n, \tau = \alpha \} \Rightarrow \{ [\tau/\alpha] C_1, [\tau/\alpha] C_2, \ldots, [\tau/\alpha] C_n \} \]
\{C_1, \ldots, C_n, \tau_1 \text{- list} = \tau_2 \text{- list} \} \Rightarrow \\
\{C_1, \ldots, C_n, \tau_1 = \tau_2 \}

\{C_1, \ldots, C_n, (\tau_1 \rightarrow \tau_2) = (\tau_1' \rightarrow \tau_2') \} \Rightarrow \\
\{C_1, \ldots, C_n, \tau_1 = \tau_1', \tau_2 = \tau_2' \}

\{C_1, \ldots, C_n, (\tau_1 \times \tau_2) = (\tau_1' \times \tau_2') \} \Rightarrow \{C_1, \ldots, C_n, \tau_1 = \tau_1', \tau_2 = \tau_2' \}

One important caveat: we will not allow constraints \( \alpha = \tau \) where \( \alpha \in \text{FV}(\tau) \). For example, we will not allow \( \alpha = \text{int} \rightarrow \alpha \).

This would lead to \( \alpha = \text{int} \rightarrow (\text{int} \rightarrow \ldots) \) a never-ending expression. (There is a way of making sense of these expressions but it is outside the scope of this class.)

Before we introduce a constraint of the form \( \alpha = \tau \) we will check if \( \alpha \) occurs in \( \tau \); this is poetically called an "occurs-check."

For example: \( \{\alpha_1 \rightarrow \alpha_2 = \text{int} \rightarrow \beta, \beta = \alpha_2 \rightarrow \alpha_2 \} \)

\Rightarrow \{\alpha_1 = \text{int}, \beta = \alpha_2, \beta = \alpha_2 \rightarrow \alpha_2 \} \Rightarrow \{\alpha_1 = \text{int}, \alpha_2 = \alpha_2 \rightarrow \alpha_2 \} \text{ OCCURS-CHECK FAILS.} \Rightarrow \text{ NOT UNIFIABLE}

We can also fail if expressions do not match so \( \tau \text{- list} = \tau_1 \rightarrow \tau_2 \) will immediately fail for example, so will \( (\tau_1 \times \tau_2) = (\tau_3 \rightarrow \tau_4) \).

The unification algorithm continues until the set of constraints is empty or none of the rules can be applied.