Solving typing constraints

$\{\alpha = int \to \beta,\ \beta = \beta_1 * bool,\ \beta_1 = int\}$ can be solved by

$$\alpha = int \to (int * bool)$$
$$\beta = int * bool \qquad \beta_1 = int$$

$\{\alpha_1 \to \alpha_2 = int \to \beta,\ \beta = int \to \alpha_1\}$

$$\alpha_1 = int \qquad \beta = int \to int = \alpha_2$$

So we can use something like Gaussian elimination. The algorithm is called <u>unification</u>

We write σ for a substitution $[\tau/\alpha]$ where τ is a type (perhaps containing type variables) and α is a type variable. We write $[\sigma]\tau$ for the effect of carrying out σ. If τ_1 & τ_2 are type expressions & σ is a substitution on <u>all</u> the type variables – so σ could look like $[\tau_1/\alpha_1, \tau_2/\alpha_2, \cdots]$ – such that

$[\sigma]\tau_1 = [\sigma]\tau_2$, where the equality sign now means identity, we say that τ_1 & τ_2 are <u>unifiable</u> & σ is the <u>unifier</u>.

How do we solve constraints? We transform a set of constraints using the following rules:

$\{C_1, C_2, \cdots C_n, int = int\} \Rightarrow \{C_1, \cdots C_n\}$

$\{C_1, C_2, \cdots C_n, bool = bool\} \Rightarrow \{C_1, \cdots, C_n\}$

$\{C_1, \cdots C_n, \alpha = \tau\} \Rightarrow \{[\tau/\alpha]C_1, [\tau/\alpha]C_2, \cdots, [\tau/\alpha]C_n\}$

$\{C_1, \cdots C_n, \tau = \alpha\} \Rightarrow \{[\tau/\alpha]C_1, [\tau/\alpha]C_2, \cdots, [\tau/\alpha]C_n\}$

$\{c_1, \cdots, c_n, \tau_1\text{-}list = \tau_2\text{-}list\} \Rightarrow$
$\{c_1, \cdots c_n, \tau_1 = \tau_2\}$ ②

$\{c_1, \cdots, c_n, (\tau_1 \to \tau_2) = (\tau_1' \to \tau_2')\} \Rightarrow$
$\{c_1, \cdots, c_n, \tau_1 = \tau_1', \tau_2 = \tau_2'\}$

$\{c_1, \cdots, c_n, (\tau_1 * \tau_2) = (\tau_1' * \tau_2')\} \Rightarrow \{c_1, \cdots, c_n, \tau_1 = \tau_1', \tau_2 = \tau_2'\}$

One important caveat: we will not allow constraints $\alpha = \tau$ where $\alpha \in FV(\tau)$. For example we will not allow

$$\alpha = int \to \alpha$$

This would lead to $\alpha = int \to (int \to \cdots)$ 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 α occurs in τ: this is poetically called an "occurs-check."

For example : $\{\alpha_1 \to \alpha_2 = int \to \beta, \beta = \alpha_2 \to \alpha_2\}$
$\Rightarrow \{\alpha_1 = int, \beta = \alpha_2, \beta = \alpha_2 \to \alpha_2\}$
$\Rightarrow \{[\alpha_2/\beta]\alpha_1 = [\alpha_2/\beta]int, [\alpha_2/\beta]\beta = [\alpha_2/\beta](\alpha_2 \to \alpha_2)\}$
$\Rightarrow \{\alpha_1 = int, \alpha_2 = \alpha_2 \to \alpha_2\}$ OCCURS-CHECK FAILS.
$\Rightarrow$ NOT UNIFIABLE

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

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