1 Two Typing Systems

We’ve discussed the rules that give us $\Gamma \vdash e : T$ in depth over the course of the semester. An expression $e$ is typable under $\Gamma$ if and only if $\exists T. (\Gamma \vdash e : T)$.

With the introduction of type variables, this typability condition must be extended: expression $e$ is typeable under $\Gamma$ if and only if $\text{solvable}(\Gamma, e) := \exists \sigma, T. (\sigma(\Gamma) \vdash \sigma(e) : T)$.

For example, $(\lambda x : A \to A . x)(\lambda y : B . y)$ cannot be typed under the original typing rules (because $A \neq B$), but $\text{solvable}((), (\lambda x : A . x)(\lambda y : B . y))$ is true (where $\sigma = \{B \mapsto A\}$).

We call this typing system the declarative system.

Constraint typing, by contrast, may be termed the algorithmic system. We define $\text{solvable}$ under constraint typing ($\text{solvable}_C$) as follows (where $C_e$ is defined by $\Gamma \vdash e : T\mid X C_e$):

$$\text{solvable}_C(\Gamma, e) := \exists \sigma . (\text{satisfies}(\sigma, C_e))$$

$satisfies$ is defined as follows:

$$\text{satisfies}(\sigma, \{s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k\}) := \forall i \leq k. (\sigma(s_i) = \sigma(t_i))$$

2 Relating the Two

We should expect, of course, that the two typing systems are equivalent. That is:

$$\forall \Gamma, e. (\text{solvable}(\Gamma, e) \Leftrightarrow \text{solvable}_C(\Gamma, e))$$

We can prove this in two steps: soundness and completeness.

For the sake of clarifying the terminology, consider what soundness and completeness would mean in the context of a theorem prover. The goal of such an algorithm would be to assert, under some given assumptions, that a suggested predicate is true. Were we to write a theorem prover, soundness would be very important to us: every assertion we make about a predicate is actually true; in other words, we never assert something that cannot be proven logically. On the other hand, completeness may be difficult to achieve in some contexts: everything that can be logically proven must be asserted as true by
In general, completeness for a theorem prover in first-order logic is impossible, because the problem is undecidable. The best we could do in many cases is to argue that our algorithm is complete under certain conditions. A claim that an algorithm is sound without making any guarantees about its completeness is a very weak assertion—the theorem prover that never asserts anything (always returning false) is still sound.

In order to talk about soundness and completeness between two systems, we need to designate a system as the “correct” or “accepted” version. Here, declarative typing is taken to be accepted. Then we define soundness as follows:

$$\forall \Gamma, e. (\text{solvable}_C(\Gamma, e) \Rightarrow \text{solvable}(\Gamma, e))$$

And completeness is:

$$\forall \Gamma, e. (\text{solvable}(\Gamma, e) \Rightarrow \text{solvable}_C(\Gamma, e))$$

We may refine this statement to say the following:

$$\forall \Gamma, e. (\text{where } \Gamma \vdash e : T', \exists \sigma, T. (\sigma(\Gamma) \vdash \sigma(e) : T) \Rightarrow \exists \sigma'. (\text{satisfies}(\sigma', C_e)))$$

For the sake of the completeness proof, we refine it further to give a stronger inductive hypothesis:

$$\forall \Gamma, e, \sigma, T. (\text{where } \Gamma \vdash e : T'\mid_X C_e. \sigma(\Gamma) \vdash \sigma(e) : T \Rightarrow \exists \sigma'. (\text{satisfies}(\sigma', C_e) \land (\text{dom}(\sigma) \cap X = \emptyset) \land (\sigma' \setminus X = \sigma)))$$

### 3 Unification

We next turn to the issue of algorithmically determining \(\text{solvable}_C(\Gamma, e)\). Our approach is to define the function \(\text{unify}(C)\) such that

$$\text{unify}(C_e) = \sigma_u \Leftrightarrow \exists \sigma. (\text{satisfies}(\sigma, C_e))$$

Given:

$$T ::= X|T \rightarrow T$$

Our initial attempt proceeds as follows:
\begin{align*}
\text{unify}(\emptyset) &= \emptyset \\
\text{unify}(t = s, C) &= \text{unify}(t, s, C) \\
\text{unify}(T, T, C) &= \text{unify}(C) \\
\text{unify}(X, T, C)[X \neq T, \sigma = \text{unify}(C), X \in \text{dom}(\sigma)] &= \text{if}(\sigma(X) = T) : \sigma \\
\text{unify}(X, T, C)[\text{otherwise}] &= \{X \mapsto T\} \cup \text{unify}(C)
\end{align*}

Class discussion ended here. A full (and correct) definition of \textit{unify}(C) is given in the text (Figure 22-2). One key concept that our attempt misses is that, if we include \{X \mapsto T\} in our result, we must replace (substitute) all occurrences of \(X\) in the remaining constraints with \(T\). For example,

\[
\text{unify}([A = B, B = A])
\]

must yield

\[
\{A \mapsto B\} \text{ or } \{B \mapsto A\}
\]

rather than

\[
\{A \mapsto B, B \mapsto A\}
\]