Chapter 6

Type Inference [Lecture 10]

Version of March 6, 2024

Summary We sketch a type inference system for the simply typed version of FUN.

6.1 An algorithm for type inference

The type systems we have seen so far guarantee that statically well-typed terms will not get stuck at runtime. However, in general it may not be obvious how to effectively *check* that an (unannotated) term really has a given type, even in the simplest FUN language without records/variants or subtyping. For example, the term $(\lambda x.\, x)\,\mathbf{true}$ has type **bool** in the empty type environment, as witnessed by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{[x \mapsto \mathbf{bool}] \vdash x : \mathbf{bool}}\,\textsc{T-Var}}{[] \vdash \lambda x.\, x : \mathbf{bool} \to \mathbf{bool}}\,\textsc{T-Lam} \qquad \dfrac{}{[] \vdash \mathbf{true} : \mathbf{bool}}\,\textsc{T-True}}{[] \vdash (\lambda x.\, x)\,\mathbf{true} : \mathbf{bool}}\,\textsc{T-App}\ .$$

But the key insight, namely that the subterm $\lambda x.\, x$ should be checked to have type $\mathbf{bool} \to \mathbf{bool}$, is not actually evident from the λ-abstraction itself; its type must be "guessed", based on how the function will subsequently be used. A problem closely related to type-checking of unannotated terms is *type inference*: given a raw term, such as $\lambda f.\, \mathbf{if}\ f\,\overline{3}\ \mathbf{then}\ \overline{4}\ \mathbf{else}\ \overline{5}$, how do we find any type for it at all (which is enough to guarantee its safety), or determine with certainty that no such type exists?

To solve these problems, we introduce an alternative, *algorithmic* notion of typing, which does not require us to guess types for subterms, but computes them explicitly – just like the evaluation judgment determines the value of a term. There are many ways of presenting this abstract algorithm, known as Hindley-Milner type inference; we choose a two-phase version based on separate notions of *constraint extraction* and *constraint solving*, because this approach scales well to more complicated systems.

We start by introducing a notion of *open types*, which may also contain occurrences of *unification variables* $\boxed{0}, \boxed{1}, \ldots$, ranged over by χ. These variables represent parts of the type that haven't been fully determined yet, but that might become known later. The grammar for open types $\hat{\tau}$ is thus:

$$\begin{array}{rcl} \chi & ::= & \boxed{i} \quad {\scriptstyle (i \in \mathbb{N})} \\ \hat{\tau} & ::= & \mathbf{int} \mid \mathbf{bool} \mid \hat{\tau}_1 \times \hat{\tau}_2 \mid \hat{\tau}_1 \to \hat{\tau}_2 \mid \chi \end{array}$$

Analogously, $\hat{\Gamma}$ maps term variables to open types. Note that every τ is also a $\hat{\tau}$. We write $UV(\hat{\tau})$ for the set of unification variables occurring in $\hat{\tau}$, defined in the obvious way by structural induction. A $\hat{\tau}$ for which $UV(\hat{\tau}) = \emptyset$ is called *ground.*

A *type substitution* $\theta = [\chi_1 \mapsto \hat{\tau}_1, ..., \chi_n \mapsto \hat{\tau}_n]$, where all the χ_j's are distinct, maps unification variables to (possibly open) types. We write $\hat{\tau}[\theta]$ for the result of applying substitution θ to the type $\hat{\tau}$. When $\hat{\tau}$ is just a variable χ, we have $\chi[\theta] = \hat{\tau}_j$ if $\chi = \chi_j$, and $\chi[\theta] = \chi$ otherwise. When $\hat{\tau}$ is not a variable, the substitution is simply applied inductively to the subcomponents of τ, if any.

For example, we have

$$(\boxed{3} \to \boxed{4} \times \mathbf{int})[\boxed{2} \mapsto \mathbf{bool}, \boxed{3} \mapsto \boxed{4}] \;=\; \boxed{4} \to \boxed{4} \times \mathbf{int}$$

Note that, since there are no binding constructs in the grammar of simple types, we don't have to worry about free vs. bound type variables, like we did for term substitution.

A substitution determined by a set of mappings is called *idempotent* if, for all i and j, χ_i does not occur in $\hat{\tau}_j$ (and thus, in particular, there are no redundant mappings $\chi_j \mapsto \chi_j$). When applying an idempotent substitution to a type, it does not matter if we substitute away the variables χ_j all at once, or one by one, in any order. All substitutions in the following will be idempotent.

A *constraint* is a relation that must hold between two open types. For simple typing, constraints will just be syntactic equations, written $\hat{\tau} \doteq \hat{\tau}'$. A *constraint system* C is then a finite sequence of such constraints:

$$C ::= \hat{\tau}_1 \doteq \hat{\tau}_1', \ldots, \hat{\tau}_n \doteq \hat{\tau}_n'$$

The empty system (i.e., with 0 equations) is written as $\cdot$. We say that a substitution θ *solves* the system C, if for every equation $\hat{\tau}_j \doteq \hat{\tau}_j'$ in C, $\hat{\tau}_j[\theta] = \hat{\tau}_j'[\theta]$. For example, the substitution $[\boxed{2} \mapsto \mathbf{bool}, \boxed{3} \mapsto \boxed{4}]$ solves the one-equation system $\boxed{3} \to \mathbf{bool} \doteq \boxed{4} \to \boxed{2}$, as does the more specific substitution $[\boxed{2} \mapsto \mathbf{bool}, \boxed{3} \mapsto \mathbf{int} \times \mathbf{int}, \boxed{4} \mapsto \mathbf{int} \times \mathbf{int}]$.

The type inference algorithm works by first generating, from the term t, a *candidate* type $\hat{\tau}$, and a collection of constraints C. Then it attempts to solve C, and if this succeeds with a substitution θ, then $\hat{\tau}[\theta]$ will be a valid type for t.

6.1.1 Generating constraints

For generating (also referred to as *extracting*) candidate types and constraints, we introduce a formal *constraint-typing judgment*, $\hat{\Gamma} \vdash^i t : \hat{\tau} \mid^{i'} C$. Here, the three arguments before the colon represent inputs to the constraint-extraction algorithm, while the three arguments after the colon represent outputs. The i and i' are natural numbers used to keep track of fresh unification-variable names: $\boxed{i}$ is the first unused variable name before generating the constraints for t, and $\boxed{i'}$ is the first unused name afterwards. The rules are shown in Figure 6.1.

Let us look closer at a couple of the CT-*???* rules; it is very instructive to compare them to the original T-*???* rules for the same constructs.

- The rule CT-Var looks up the type of the variable x in $\hat{\Gamma}$, just like T-Var did before. It does not introduce any new unification variables (so $i' = i$) nor constraints (so $C = \cdot$). The rules for numeric and boolean constants are analogous.

Judgment $\boxed{\hat\Gamma \vdash^i t : \hat\tau \mid^{i'} C}$

$$\textsc{CT-Var} : \frac{}{\hat\Gamma \vdash^i x : \hat\tau \mid^i \cdot}\ (\hat\Gamma(x) = \hat\tau) \qquad \textsc{CT-Num} : \frac{}{\hat\Gamma \vdash^i \overline{n} : \mathbf{int} \mid^i \cdot}$$

$$\textsc{CT-True} : \frac{}{\hat\Gamma \vdash^i \mathbf{true} : \mathbf{bool} \mid^i \cdot} \qquad \textsc{CT-False} : \frac{}{\hat\Gamma \vdash^i \mathbf{false} : \mathbf{bool} \mid^i \cdot}$$

$$\textsc{CT-Plus} : \frac{\hat\Gamma \vdash^i t_0 : \hat\tau_0 \mid^{i''} C_0 \qquad \hat\Gamma \vdash^{i''} t_1 : \hat\tau_1 \mid^{i'} C_1}{\hat\Gamma \vdash^i t_0 + t_1 : \mathbf{int} \mid^{i'} C_0, C_1, \hat\tau_0 \doteq \mathbf{int}, \hat\tau_1 \doteq \mathbf{int}}$$

$$\textsc{CT-Leq} : \frac{\hat\Gamma \vdash^i t_0 : \hat\tau_0 \mid^{i''} C_0 \qquad \hat\Gamma \vdash^{i''} t_1 : \hat\tau_1 \mid^{i'} C_1}{\hat\Gamma \vdash^i t_0 \leq t_1 : \mathbf{bool} \mid^{i'} C_0, C_1, \hat\tau_0 \doteq \mathbf{int}, \hat\tau_1 \doteq \mathbf{int}}$$

$$\textsc{CT-If} : \frac{\hat\Gamma \vdash^i t_0 : \hat\tau_0 \mid^{i''} C_0 \qquad \hat\Gamma \vdash^{i''} t_1 : \hat\tau_1 \mid^{i'''} C_1 \qquad \hat\Gamma \vdash^{i'''} t_2 : \hat\tau_2 \mid^{i'} C_2}{\hat\Gamma \vdash^i \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \hat\tau_1 \mid^{i'} C_0, C_1, C_2, \hat\tau_0 \doteq \mathbf{bool}, \hat\tau_1 \doteq \hat\tau_2}$$

$$\textsc{CT-Pair} : \frac{\hat\Gamma \vdash^i t_1 : \hat\tau_1 \mid^{i''} C_1 \qquad \hat\Gamma \vdash^{i''} t_2 : \hat\tau_2 \mid^{i'} C_2}{\hat\Gamma \vdash^i (t_1, t_2) : \hat\tau_1 \times \hat\tau_2 \mid^{i'} C_1, C_2}$$

$$\textsc{CT-Fst} : \frac{\hat\Gamma \vdash^i t_0 : \hat\tau_0 \mid^{i'} C_0}{\hat\Gamma \vdash^i \mathbf{fst}(t_0) : \boxed{i'} \mid^{i'+2} C_0, \hat\tau_0 \doteq \boxed{i'} \times \boxed{i'+1}}$$

$$\textsc{CT-Snd} : \frac{\hat\Gamma \vdash^i t_0 : \hat\tau_0 \mid^{i'} C_0}{\hat\Gamma \vdash^i \mathbf{snd}(t_0) : \boxed{i'+1} \mid^{i'+2} C_0, \hat\tau_0 \doteq \boxed{i'} \times \boxed{i'+1}}$$

$$\textsc{CT-Lam} : \frac{\hat\Gamma[x \mapsto \boxed{i}] \vdash^{i+1} t_0 : \hat\tau_0 \mid^{i'} C_0}{\hat\Gamma \vdash^i \lambda x.\, t_0 : \boxed{i} \to \hat\tau_0 \mid^{i'} C_0}$$

$$\textsc{CT-App} : \frac{\hat\Gamma \vdash^i t_1 : \hat\tau_1 \mid^{i''} C_1 \qquad \hat\Gamma \vdash^{i''} t_2 : \hat\tau_2 \mid^{i'} C_2}{\hat\Gamma \vdash^i t_1\, t_2 : \boxed{i'} \mid^{i'+1} C_1, C_2, \hat\tau_1 \doteq \hat\tau_2 \to \boxed{i'}}$$

$$\textsc{CT-Let} : \frac{\hat\Gamma \vdash^i t_1 : \hat\tau_1 \mid^{i''} C_1 \qquad \hat\Gamma[x \mapsto \hat\tau_1] \vdash^{i''} t_2 : \hat\tau_2 \mid^{i'} C_2}{\hat\Gamma \vdash^i \mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 : \hat\tau_2 \mid^{i'} C_1, C_2}$$

$$\textsc{CT-Rec} : \frac{\hat\Gamma[x \mapsto \boxed{i}] \vdash^{i+1} t_0 : \hat\tau_0 \mid^{i'} C_0}{\hat\Gamma \vdash^i \mathbf{rec}\ x.\, t_0 : \hat\tau_0 \mid^{i'} C_0, \boxed{i} \doteq \hat\tau_0}$$

Figure 6.1: Constraint-generating typing rules

- CT-Plus says that the result of the addition will have type **int**, but introduces the constraints that the two summands must also both have type **int**. Note how the constraints generated for t_0 and t_1 are joined together, and how the next-unused-variable index is threaded through the premises like a state.
- Skipping down to CT-Lam, we see that, instead of picking a specific type for the bound variable x in the extended typing environment, it simply introduces a new, unconstrained unification variable $\boxed{i}$ to stand for that type. However, either the body t_0 or other parts of the program may subsequently generate constraints involving $\boxed{i}$, so it may not stay unconstrained forever. Note also how the i is incremented before t_0 is processed, so that we do not generate $\boxed{i}$ as a fresh variable again.
- Conversely, CT-App generates candidate types and constraints for the function and argument subterms t_1 and t_2, and then adds the constraint that the type $\hat{\tau}_1$ of t_1 must be a function type, in which the argument type is the same as the type $\hat{\tau}_2$ of t_2, and the result type is the type of the whole application, represented by the new unification variable $\boxed{i'}$. The next free variable index is then $i' + 1$.
- Finally, CT-Rec is similar to CT-Lam, but generates the constraint that the type used for x must equal the type of the body t_0 of the **rec**-term.

Unlike the original typing rules, the constraint-typing system is completely deterministic, in the sense that $\hat{\Gamma}$, i, and t (i.e., everything before the ":" in the judgment) uniquely determine $\hat{\tau}$, i', and C (i.e., everything after the ":"). That is, we can show:

Lemma 6.1 *If* $\hat{\Gamma} \vdash^i t : \hat{\tau} \mid^{i'} C$ *and* $\hat{\Gamma} \vdash^i t : \hat{\tau}' \mid^{i''} C'$, *then* $\hat{\tau} = \hat{\tau}'$, $i' = i''$, *and* $C = C'$.

Proof. Simple structural induction on t.

Note also, how all premises in the constraint-typing rules are of the form $t_i : \hat{\tau}_i$, where the shape of $\hat{\tau}_i$ is not further restricted by the rule itself. All equations between types are put into C instead, to be resolved later. The only situation in which the constraint generator can fail entirely is thus when it encounters a variable that is not assigned a type in $\hat{\Gamma}$, so that the side condition in CT-Var fails. Barring that, the judgment will always succeed in finding a candidate type and associated constraints. However, even though the extraction always succeeds, the constraints may still prove unsolvable. For example, we can derive

$$[] \vdash^0 \overline{3} + \mathbf{true} : \mathbf{int} \mid^0 (\mathbf{int} \doteq \mathbf{int}, \mathbf{bool} \doteq \mathbf{int}),$$

but the second constraint in the generated system evidently cannot be solved by any θ.

Example. Let us look at a larger example of the algorithm in action. We want to find the type and constraints for $t \equiv \mathbf{rec}\ f.\, \lambda x.\, \lambda y.\, \mathbf{if}\ \overline{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \overline{1})\, y$, starting in the empty type environment.

The trace of the algorithm, considered as a functional program, is shown in Figure 6.2. Each line labelled with $\Rightarrow$ represents an invocation of the type/constraint extractor with a given type environment, term, and starting $\boxed{i}$-index; the matching $\Leftarrow$ line, at the same level of indentation, also shows the extracted type, next free index, and generated constraints.

$$
\begin{array}{l}
\Rightarrow [\,] \vdash^0 \mathbf{rec}\ f.\, \lambda x.\, \lambda y.\, \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : ? \\
\quad \Rightarrow [f \mapsto \boxed{0}] \vdash^1 \lambda x.\, \lambda y.\, \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : ? \\
\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}] \vdash^2 \lambda y.\, \mathbf{if}\ 0 \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : ? \\
\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : ? \\
\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{0} \leq x : ? \\
\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{0} : ? \\
\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{0} : \mathbf{int} \mid^3 \cdot \\
\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x : ? \\
\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x : \boxed{1} \mid^3 \cdot \\
\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{0} \leq x : \mathbf{bool} \mid^3 \ldots, \mathbf{int} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int} \\
\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 y : ? \\
\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 y : \boxed{2} \mid^3 \cdot \\
\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f\,(x + \bar{1})\, y : ? \\
\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f\,(x + \bar{1}) : ? \\
\quad\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f : ? \\
\quad\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f : \boxed{0} \mid^3 \cdot \\
\quad\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x + \bar{1} : ? \\
\quad\quad\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x : ? \\
\quad\quad\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x : \boxed{1} \mid^3 \cdot \\
\quad\quad\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{1} : ? \\
\quad\quad\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 \bar{1} : \mathbf{int} \mid^3 \cdot \\
\quad\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 x + \bar{1} : \mathbf{int} \mid^3 \ldots, \boxed{1} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int} \\
\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f\,(x + \bar{1}) : \boxed{3} \mid^4 \ldots, \boxed{0} \doteq \mathbf{int} \to \boxed{3} \\
\quad\quad\quad\quad\quad \Rightarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^4 y : ? \\
\quad\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^4 y : \boxed{2} \mid^4 \cdot \\
\quad\quad\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}, y \mapsto \boxed{2}] \vdash^3 f\,(x + \bar{1})\, y : \boxed{4} \mid^5 \ldots, \boxed{3} \doteq \boxed{2} \to \boxed{4} \\
\quad\quad\quad \Leftarrow [\cdots] \vdash^3 \mathbf{if}\ 0 \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : \boxed{2} \mid^5 \ldots, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4} \\
\quad\quad \Leftarrow [f \mapsto \boxed{0}, x \mapsto \boxed{1}] \vdash^2 \lambda y.\, \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : \boxed{2} \to \boxed{2} \mid^5 \ldots \\
\quad \Leftarrow [f \mapsto \boxed{0}] \vdash^1 \lambda x.\, \lambda y.\, \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : \boxed{1} \to \boxed{2} \to \boxed{2} \mid^5 \ldots \\
\Leftarrow [\,] \vdash^0 \mathbf{rec}\ f.\, \lambda x.\, \lambda y.\, \mathbf{if}\ \bar{0} \leq x\ \mathbf{then}\ y\ \mathbf{else}\ f\,(x + \bar{1})\, y : \boxed{1} \to \boxed{2} \to \boxed{2} \mid^5 \ldots, \boxed{0} \doteq \boxed{1} \to \boxed{2} \to \boxed{2}
\end{array}
$$

Figure 6.2: Trace of the constraint-generation algorithm

For space reasons, since all constraints are ultimately accumulated into one big system, we only display the *new* constraints generated by each rule instance, with "…" representing all the constraints generated by the premises of the rule. Thus, each constraint that is introduced by the rules is written out exactly once in the trace.

Ultimately, the constraint-typing system finds that t has candidate type $\boxed{1} \to \boxed{2} \to \boxed{2}$, subject to the following constraints: $\mathbf{int} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int}, \boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \boxed{0} \doteq \boxed{1} \to \boxed{2} \to \boxed{2}$.

6.1.2 Solving constraints

Once the constraints have been generated, we still need to solve them, or detect that they are unsolvable. We will express this part of the algorithm, commonly known as *unification*, as a transition judgment that gradually reduces the constraint system into a form that makes it obvious what the solution is, or that there is no solution. To

Judgment $\boxed{C \rightsquigarrow C'}$ (Since all rules are axioms, we omit the overbars).

$$\begin{array}{rl}
\text{CS-DecII}: & \mathbf{int} \doteq \mathbf{int}, C \rightsquigarrow C \\
\text{CS-DecBB}: & \mathbf{bool} \doteq \mathbf{bool}, C \rightsquigarrow C \\
\text{CS-DecPP}: & \hat{\tau}_1 \times \hat{\tau}_2 \doteq \hat{\tau}'_1 \times \hat{\tau}'_2, C \rightsquigarrow \hat{\tau}_1 \doteq \hat{\tau}'_1, \hat{\tau}_2 \doteq \hat{\tau}'_2, C \\
\text{CS-DecFF}: & \hat{\tau}_1 \to \hat{\tau}_2 \doteq \hat{\tau}'_1 \to \hat{\tau}'_2, C \rightsquigarrow \hat{\tau}_1 \doteq \hat{\tau}'_1, \hat{\tau}_2 \doteq \hat{\tau}'_2, C
\end{array}$$

$$\begin{array}{rl}
\text{CS-ClashIB}: & \mathbf{int} \doteq \mathbf{bool}, C \rightsquigarrow \mathbf{fail} \\
\text{CS-ClashIP}: & \mathbf{int} \doteq \hat{\tau}_1 \times \hat{\tau}_2, C \rightsquigarrow \mathbf{fail} \\
\text{CS-ClashIF}: & \mathbf{int} \doteq \hat{\tau}_1 \to \hat{\tau}_2, C \rightsquigarrow \mathbf{fail}
\end{array}$$

(+ analogous rules for the other 9 clashing combinations)

$$\begin{array}{rl}
\text{CS-Triv}: & \chi \doteq \chi, C \rightsquigarrow C \\
\\
\text{CS-OccL}: & \chi \doteq \hat{\tau}, C \rightsquigarrow \mathbf{fail} \quad (\chi \in UV(\hat{\tau}) \land \chi \neq \hat{\tau}) \\
\text{CS-OccR}: & \hat{\tau} \doteq \chi, C \rightsquigarrow \mathbf{fail} \quad (\chi \in UV(\hat{\tau}) \land \chi \neq \hat{\tau}) \\
\\
\text{CS-ElimL}: & \chi \doteq \hat{\tau}, C \rightsquigarrow C[\hat{\tau}/\chi], \chi \doteq^{\surd} \hat{\tau} \quad (\chi \notin UV(\hat{\tau})) \\
\text{CS-ElimR}: & \hat{\tau} \doteq \chi, C \rightsquigarrow C[\hat{\tau}/\chi], \chi \doteq^{\surd} \hat{\tau} \quad (\chi \notin UV(\hat{\tau}))
\end{array}$$

Figure 6.3: Constraint-simplification rules

accommodate the latter alternative, we also allow a constraint system C to be the single atom **fail**, representing a failure.

We say that an equation in C is *solved* if it is of the form $\chi \doteq \hat{\tau}$, where χ does not occur anywhere else in C, including in $\hat{\tau}$. We mark such equations with a checkmark, $\doteq^{\surd}$, and group them at the end of the system. A constraint system is said to be in *normal form* if it is either **fail** or consists of only solved equations $(\chi_1 \doteq^{\surd} \hat{\tau}_1, \ldots, \chi_n \doteq^{\surd} \hat{\tau}_n)$. In the latter case, we can immediately read off the solving substitution $[\chi_1 \mapsto \hat{\tau}_1, \ldots, \chi_n \mapsto \hat{\tau}_n]$.

The transition rules are shown in Figure 6.3. Their intuitive justification is that each transition simplifies the system, without changing the set of its solutions, i.e., whenever $C \rightsquigarrow C'$, then θ solves C iff θ solves C'. Thus, if $C \rightsquigarrow^* C'$, where C' contains only solved equations, then the evident corresponding substitution also solves the original C; and conversely, if $C \rightsquigarrow^*$ **fail**, then the original C was also unsolvable.

The first group of rules, CS-Dec*??*, are called *decomposition rules.* Constraints of the form **int** $\doteq$ **int** are solved by any substitution, so they can just be thrown out. More interestingly, by the inductive definition of substitution application, we have that $(\hat{\tau}_1 \to \hat{\tau}_2)[\theta] = (\hat{\tau}'_1 \to \hat{\tau}'_2)[\theta]$ precisely when $\hat{\tau}_1[\theta] = \hat{\tau}'_1[\theta]$ and $\hat{\tau}_2[\theta] = \hat{\tau}'_2[\theta]$. Thus we can replace an equation $\hat{\tau}_1 \to \hat{\tau}_2 \doteq \hat{\tau}'_1 \to \hat{\tau}'_2$ in C with the two simpler equations $\hat{\tau}_1 \doteq \hat{\tau}'_1$ and $\hat{\tau}_2 \doteq \hat{\tau}'_2$, without changing the possible solutions of the constraint system.

The CS-Clash*??* rules identify constraints that are patently unsolvable. For example, for no θ can $\mathbf{int}[\theta] = (\hat{\tau}_1 \to \hat{\tau}_2)[\theta]$, so any constraint set containing $\mathbf{int} \doteq (\hat{\tau}_1 \to \hat{\tau}_2)$ can be immediately reduced to failure by CS-ClashIF.

The CS-Triv rule expresses that, for any θ, $\chi[\theta] = \chi[\theta]$, so a constraint of the form $\chi \doteq \chi$ does not restrict the solution θ at all and can be discarded, just like **int** $\doteq$ **int**. On the other hand, if we encounter a constraint like $\chi \doteq \mathbf{int} \to \chi$, then clearly for no θ will we get $\chi[\theta] = (\mathbf{int} \to \chi)[\theta]$, because the RHS of the equation is always strictly larger than the LHS, no matter what θ substitutes for χ. Accordingly, the rules CS-Occ*?* say that whenever we encounter a constraint equating a variable χ to a $\hat{\tau}$ in which χ also occurs (except as all of $\hat{\tau}$), that constraint is unsolvable, and thus so is the whole set.

Finally, the rules CS-Elim*?* say that if we encounter an equation $\chi \doteq \hat{\tau}$, where χ does not occur in $\hat{\tau}$, we can eliminate the variable χ by substituting $\hat{\tau}$ for χ in all remaining equations (including solved ones). We still retain $\chi \doteq \hat{\tau}$ as part of the constraint system, but we put it at the end and mark it solved, since we will not need to consider it again: after the substitution, there are no other occurrences of χ left in the constraint system, so a solved equation can never become un-solved by later substitutions. Note that an equation $\boxed{i} \doteq \boxed{i'}$, where $i \neq i'$, can reduce by either of two the CS-Elim*?* rules; it does not matter for the correctness of the algorithm which one we use. If we want a completely deterministic rule set, we may add the additional side condition in CS-ElimR that $\hat{\tau}$ must not be another unification variable (because CS-ElimL would already cover that case).

Example Let us return to the constraint set generated by the example at the end of the previous section. In Figure 6.4, we show how the constraints are gradually reduced to a solved system. In particular, we can read off the solving substitution $\theta = [\boxed{1} \mapsto \mathbf{int}, \boxed{0} \mapsto \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \mapsto \boxed{4} \to \boxed{4}, \boxed{2} \mapsto \boxed{4}]$. Applying this to the candidate type $\hat{\tau} = \boxed{1} \to \boxed{2} \to \boxed{2}$, we get $\hat{\tau}[\theta] = \mathbf{int} \to \boxed{4} \to \boxed{4}$, which says that any further substitution of a ground type for $\boxed{4}$ gives a valid type of the original term. (In a polymorphic type system, we could instead assign t the type *schema* $\forall \alpha.\, \mathbf{int} \to \alpha \to \alpha$.)

6.1.3 Type inference in practice

The abstract algorithm can be programmed directly (see Exercise 6.1). In practice, however, for efficiency reasons, it is usually implemented with a few adaptations:

- Instead of generating all constraints first and then solving them, most type inference algorithms interleave constraint generation and constraint simplification, so that typing failures are detected early, and so that unsolved equations are not kept around for longer than they need to be.
- Instead of expressing variable elimination by actually substituting away the variable in the remaining constraints, the algorithm maintains a *unification heap* ρ, which maps unification variables to open types. Then in CS-Elim, ρ is simply extended with the new binding for χ. Conversely, when a unification variable is encountered in an open type, it is always looked up in ρ first, to see if if has acquired a binding already. However, unlike a substitution application $\chi[\theta]$, if the heap maps χ to another variable χ', we have to look up χ' again, until we reach either a non-variable type, or a variable that does not (yet) have a binding in ρ.
- The unification heap ρ is not actually maintained as an explicit function or other immutable data structure, but as a store with destructively updated cells. In ML/F#, this is typically expressed using the **ref**-construct.

$\mathbf{int} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int}, \boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4},$
$\boxed{0} \doteq \boxed{1} \to \boxed{2} \to \boxed{2}$

$\quad\leadsto$ (by CS-DecII)

$\boxed{1} \doteq \mathbf{int}, \boxed{1} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int}, \boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4},$
$\boxed{0} \doteq \boxed{1} \to \boxed{2} \to \boxed{2}$

$\quad\leadsto$ (by CS-ElimL)

$(\boxed{1} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int}, \boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \boxed{0} \doteq \boxed{1} \to \boxed{2} \to \boxed{2})$
$[\mathbf{int}/\boxed{1}], \boxed{1} \doteq^{\surd} \mathbf{int}$

$\quad =$

$\mathbf{int} \doteq \mathbf{int}, \mathbf{int} \doteq \mathbf{int}, \boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \boxed{0} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2},$
$\boxed{1} \doteq^{\surd} \mathbf{int}$

$\quad\leadsto^*$ (by CS-DecII twice)

$\boxed{0} \doteq \mathbf{int} \to \boxed{3}, \boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \boxed{0} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}$

$\quad\leadsto$ (by CS-ElimL)

$(\boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \boxed{0} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int})[(\mathbf{int} \to \boxed{3})/\boxed{0}],$
$\boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{3}$

$\quad =$

$\boxed{3} \doteq \boxed{2} \to \boxed{4}, \mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \mathbf{int} \to \boxed{3} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{3}$

$\quad\leadsto$ (by CS-ElimL)

$(\mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \mathbf{int} \to \boxed{3} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{3})[\boxed{2} \to \boxed{4}/\boxed{3}],$
$\boxed{3} \doteq^{\surd} \boxed{2} \to \boxed{4}$

$\quad =$

$\mathbf{bool} \doteq \mathbf{bool}, \boxed{2} \doteq \boxed{4}, \mathbf{int} \to \boxed{2} \to \boxed{4} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{2} \to \boxed{4},$
$\boxed{3} \doteq^{\surd} \boxed{2} \to \boxed{4}$

$\quad\leadsto$ (by CS-DecBB)

$\boxed{2} \doteq \boxed{4}, \mathbf{int} \to \boxed{2} \to \boxed{4} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{2} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{2} \to \boxed{4}$

$\quad\leadsto$ (by CS-ElimL)

$(\mathbf{int} \to \boxed{2} \to \boxed{4} \doteq \mathbf{int} \to \boxed{2} \to \boxed{2}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{2} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{2} \to \boxed{4})$
$[\boxed{4}/\boxed{2}], \boxed{2} \doteq^{\surd} \boxed{4},$

$\quad =$

$\mathbf{int} \to \boxed{4} \to \boxed{4} \doteq \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{4} \to \boxed{4}, \boxed{2} \doteq^{\surd} \boxed{4}$

$\quad\leadsto$ (by CS-DecFF)

$\mathbf{int} \doteq \mathbf{int}, \boxed{4} \to \boxed{4} \doteq \boxed{4} \to \boxed{4}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{4} \to \boxed{4}, \boxed{2} \doteq^{\surd} \boxed{4}$

$\quad\leadsto$ (by CS-DecII)

$\boxed{4} \to \boxed{4} \doteq \boxed{4} \to \boxed{4}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{4} \to \boxed{4}, \boxed{2} \doteq^{\surd} \boxed{4}$

$\quad\leadsto$ (by CS-DecFF)

$\boxed{4} \doteq \boxed{4}, \boxed{4} \doteq \boxed{4}, \boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{4} \to \boxed{4}, \boxed{2} \doteq^{\surd} \boxed{4}$

$\quad\leadsto^*$ (by CS-Triv twice)

$\boxed{1} \doteq^{\surd} \mathbf{int}, \boxed{0} \doteq^{\surd} \mathbf{int} \to \boxed{4} \to \boxed{4}, \boxed{3} \doteq^{\surd} \boxed{4} \to \boxed{4}, \boxed{2} \doteq^{\surd} \boxed{4}$

Figure 6.4: Trace of the constraint simplification algorithm

6.2 Correctness of type inference

[**Note:** This section is just a sketch, and may be buggy, though the statements are believed to be "morally correct". The correctness lemmas and theorem below are *not* part of the syllabus for *Semantics and Types 2023/24*.]

Lemma 6.2 (Soundness of constraint extractor) *If* $\hat{\Gamma} \vdash^{i} t : \hat{\tau} \mid^{i'} C$*, and* θ *is a grounding substitution (i.e., such that* $\chi[\theta]$ *is ground for every* $\chi \in UV(\hat{\Gamma}) \cup \{\boxed{i}, ..., \boxed{i'-1}\}$*) that solves* C*, then* $\hat{\Gamma}[\theta] \vdash t : \hat{\tau}[\theta]$*.*

Lemma 6.3 (Completeness of constraint extractor) *If* $\Gamma \vdash t : \tau$*,* $\hat{\Gamma}[\theta] = \Gamma$*, and all* $\boxed{j} \in UV(\hat{\Gamma})$ *satisfy* $j < i$*, then* $\Gamma \vdash^{i} t : \hat{\tau} \mid^{i'} C$ *for some* $\hat{\tau}$*,* i'*, and* C*; and there exists a substitution* θ' *with* $\operatorname{dom}\theta' \subseteq \{\boxed{i}, ..., \boxed{i'-1}\}$ *that solves* C*, and such that* $\hat{\tau}[\theta][\theta'] = \tau$*.*

Lemma 6.4 (Partial correctness of constraint solver) *If* $C \rightsquigarrow C'$*, then* θ *solves* C *if and only if* θ *solves* C'*.*

Lemma 6.5 (Termination of constraint solver) *For any* C *(with any solved equations grouped at the end), either* $C \rightsquigarrow^{*}$ **fail***, or* $C \rightsquigarrow^{*} C'$ *for some* C' *containing only solved equations.*

Putting it all together, we can then formalize that our two-phase algorithm finds all valid typings for any closed term:

Theorem 6.6 (Correctness of constraint-typing) *Let* t *be a closed term. Then, for some* $\hat{\tau}$*,* i' *and* C*, we derive* $[] \vdash^{0} t : \hat{\tau} \mid^{i'} C$*; and for some* C' *in normal form (i.e., solved or* **fail***),* $C \rightsquigarrow^{*} C'$*. And then, for any* τ*,* $[] \vdash t : \tau$ *iff* $\tau = \hat{\tau}[\chi_1 \mapsto \hat{\tau}_1, ..., \chi_n \mapsto \hat{\tau}_n][\theta']$*, where* $C' = (\chi_1 \doteq^{\surd} \hat{\tau}_1, ..., \chi_n \doteq^{\surd} \hat{\tau}_n)$*, and* θ' *is some grounding substitution for* $\boxed{0}, ..., \boxed{i'-1}$*.*

6.3 Exercises

6.1. Implement the constraint generator and the constraint solver in ML/F# or Haskell.