CS 185: Lecture Notes on Simple Types.

Syntax. The syntax of simple types is given by

\[(\text{type}) ::= (\text{base}) | (\text{type} \rightarrow \text{type})\]

where \((\text{base})\) is any non-empty set of base types (for example, \text{int} or char). We use \(T\) as a meta-variable for types. The intuition is that \(T_1 \rightarrow T_2\) is supposed to be the type for functions with domain \(T_1\) and range \(T_2\).

Typing. In this setting, expressions (elements of \((\text{exp})\)) are often called terms, to distinguish them from types. We use meta-variable \(t\) for terms. We are interested in when a lambda term be assigned a simple type. For example, \(\lambda x.x\) can be assigned any simple type of the form \(T \rightarrow T\), since the identity function can be considered to have domain \(T\) and range \(T\) for any simple type \(T\). The following rules inductively define the simple type assignment relation. In the notation \(\Gamma \vdash t : T\), \(t\) is a lambda term to be assigned simple type \(T\), and \(\Gamma\) is a \((\text{context})\) assigning simple types to the free variables of \(t\):

\[(\text{context}) ::= \cdot | ([\text{context}], \langle \text{var} \rangle : \langle \text{type} \rangle)\]

The context \(\cdot\) is the empty context. It is common to view contexts as functions from variables to simple types. So in the first rule below, the notation \(\Gamma(x) = T\) is used to mean that the result of looking up the type for variable \(x\) in context \(\Gamma\) is \(T\) (i.e., the function \(\Gamma\) returns type \(T\) for \(x\)).

\[
\begin{align*}
\Gamma(x) &= T \\
\Gamma \vdash x : T \\
\Gamma \vdash t_1 : T_2 \rightarrow T_1 & \quad \Gamma \vdash t_2 : T_2 & \quad \Gamma, x : T_1 \vdash t : T_2 \\
\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2
\end{align*}
\]

There are two ways we might try to use these rules as an algorithm, both of which end up being infinitarily non-deterministic (and hence unusable):

1. **Type checking.** The most natural interpretation, given the above remarks, is to the one taking the judgment \(\Gamma \vdash t : T\) as specifying \(\Gamma\), \(t\), and \(T\) as inputs. So the judgment expresses that we check whether \(t\) can be assigned simple type \(T\) in context \(\Gamma\). The problem with this reading is that when we apply the application rule, we must non-deterministically guess type \(T_2\) as we pass from its conclusion to its premises. There are an infinite number of choices, since there are infinitely many simple types. Note, however, that the other rules can both be executed deterministically.

2. **Type computation.** We could also interpret \(\Gamma \vdash t : T\) as specifying \(\Gamma\) and \(t\) as inputs, and \(T\) as output. In this case, the judgment expresses the idea that simple type \(T\) can be computed for \(t\) in context \(\Gamma\). The application rule is completely deterministic on this reading: if we have computed type \(T_2 \rightarrow T_1\) for \(t_1\) and type \(T_2\) for \(t_2\), then we compute type \(T_1\) for the application of \(t_1\) to \(t_2\). The problem with the type computation reading shows up in the rule for typing \(\lambda\)-abstractions. There, we must non-deterministically guess the type \(T_1\) to give to \(x\) in the extended context in the premise of the rule. So once again, the rules are infinitarily non-deterministic.

Before we deal further with this issue, we verify that our typing relation is sound, in the following sense.

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

The proof of this relies on a substitution lemma, which shows how typing commutes with substitution. With type preservation established, we can now consider different ways to obtain a deterministic algorithm for typing:

1. **Annotated applications for type checking.** If we wish to use the typing rules for type checking, then we can annotate applications to remove the non-determinism in the application rule (described above). We may add annotations to applications by the syntax of \(\lambda\)-terms \(t\):

\[(\text{exp}) ::= \langle \text{var} \rangle | (\text{exp}) (\text{exp}) | [(\text{type})] | \lambda \langle \text{var} \rangle. (\text{exp})\]
The typing rules above are then modified as follows (note that only the application rule has changed):

\[
\begin{align*}
\Gamma(x) &= T \\
\Gamma \vdash x : \bar{T} &\quad \Gamma \vdash t_1 : T_2 \rightarrow T_1 \quad \Gamma \vdash t_2 : T_2 \\
&\quad \Gamma, x : T_3 \vdash t : T_2 \\
&\quad \Gamma \vdash \lambda x.t : T_1 \rightarrow T_2
\end{align*}
\]

This approach is admittedly not commonly used in practice, though it is perfectly adequate theoretically.

2. **Annotated abstractions for type computation.** More commonly, if we wish to use the typing rules for type computation, then we can annotate \(\lambda\)-abstractions to remove the non-determinism in the abstraction rule:

\[
\langle \exp \rangle ::= \langle \var \rangle \mid \langle \exp \rangle \langle \exp \rangle \mid \lambda \langle \var \rangle : \langle \type \rangle
\]

The typing rules above are then modified as follows (only the \(\lambda\)-abstraction rule has changed):

\[
\begin{align*}
\Gamma(x) &= T \\
\Gamma \vdash x : \bar{T} &\quad \Gamma \vdash t_1 : T_2 \rightarrow T_1 \quad \Gamma \vdash t_2 : T_2 \\
&\quad \Gamma, x : T_3 \vdash t : T_2 \\
&\quad \Gamma \vdash \lambda x.t : T_1 \rightarrow T_2
\end{align*}
\]

3. **Constraint generation.** Rather than add annotations, we modify our typing 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 \(\langle \meta-var \rangle\), defined by the following syntax:

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

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).

\[
\begin{align*}
\Gamma(x) &= T \\
\Gamma \vdash x : T > \cdot &\quad \Gamma \vdash t_1 : T_1 > C_1 \quad \Gamma \vdash t_2 : T_2 > C_2 \\
&\quad \Gamma, x : T_3 \vdash t : T > C \\
&\quad \Gamma \vdash \lambda x.t : T \rightarrow \bar{T} \\
&\quad \Gamma \vdash \lambda x.t : X \rightarrow \bar{T} > \bar{C}
\end{align*}
\]

**Next time:** constraint solving, strong normalization of simply typable terms.