4. Type Checking and Type Inference

Complex Haskell $\Rightarrow$ Simple Haskell $\Rightarrow$ Lambda Calculus

Then one should first do type-checking for the obtained $\lambda$-term. If it is well typed, it should then be evaluated.

4.1 Type Schemas and Type Assumptions

Main problem: polymorphism

An expression like Nil may have several types:

List $a$, List Int, List Bool, ...

We want to compute the most general type of any $\lambda$-term (e.g., List $a$). Here, the type variable $a$ stands for any possible type. To make this clearer, we now use type schemas: $\forall a. \text{List } a$

For pre-defined functions and constructors (e.g., constants $\epsilon$ of the $\lambda$-calculus), we need to know their types before we start
type-checking.

This information is stored in a

\textit{type assumption} (\textit{\textit{\equiv}} “environment” when
defining the semantics of
Haskell)

The \textit{type assumption} assigns a type schema to
every constant (\textit{\textit{\textit{\textit{\equiv}}} from } \mathcal{E}) and every variable
(\textit{\textit{\textit{\textit{\textit{\equiv}}} from } \mathcal{V}) of the \textit{\textit{\textit{\lambda}}} -calculus.

We start with an initial \textit{type assumption} \( A_0 \):

\begin{align*}
A_0 (\text{not}) &= \text{Bool} \to \text{Bool} \\
A_0 (+) &= \text{Int} \to \text{Int} \to \text{Int} \\
A_0 (5) &= \text{Int} \\
A_0 (\times) &= \forall a. a \ (\text{for any } x \in \mathcal{V})
\end{align*}

For data \textit{List} \( a = \text{Nil} | \text{Cons } a \ (\text{List } a) \):

\begin{align*}
A_0 (\text{Nil}) &= \forall a. \text{List } a \\
A_0 (\text{Cons}) &= \forall a. a \to (\text{List } a) \to (\text{List } a)
\end{align*}
Type schemas are built according to the following grammar:

\[
\text{typeschema} \rightarrow \text{tyconstr typeschema}_1 \ldots \text{typeschema}_n, \ n \geq 0 \\
| \ (\text{typeschema}_1 \rightarrow \text{typeschema}_2) \\
| \ (\text{typeschema}_1, \ldots, \text{typeschema}_n), \ n \geq 0 \\
| \ \text{var} \\
| \ \forall \text{var} : \text{typeschema}
\]

For a typeschema \( T \) with free variables \( a_1, \ldots, a_n \), we write \( \forall T \) for \( \forall a_1 \ldots \forall a_n. \ T \).

A type assumption \( A \) is a (possibly partial) function from \( \mathcal{V} \cup \mathcal{E} \) to the set of type schemas.

A type assumption \( A \) with \( A(x_i) = T_i \) for \( 1 \leq i \leq n \) which is undefined on other arguments is also written \( A = \{ x_1 : T_1, \ldots, x_n : T_n \} \).

The initial type assumption \( A_0 \) is defined on Slide 59.
Here, we assume that Constr is a user-defined data constructor introduced by:

```
data tyconstr a_1 \ldots a_m = \ldots (Constr \; type_1 \ldots type_n) \ldots
```

For two type assumptions $A$ and $A'$, we define $A + A'$ as:

$$ (A + A')(x) = \begin{cases} A'(x), & \text{if } A'(x) \text{ is defined} \\ A(x), & \text{otherwise} \end{cases} $$