4. Type Checking and Type Inference

Complex Haskell ⇒ Simple Haskell ⇒ Lambda Calculus

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

4.1. Type Schemas + 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 λ-term (e.g. List a). Here, the type variable a stands for any possible type. To make this clearer, we now use

<u>type schemas</u> : $\forall a.\ \text{List } a$

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

$\Rightarrow$ This information is stored in a <u>type assumption</u> ($\hat{=}$ "environment" when defining the semantics of Haskell)

The <u>type assumption</u> assigns a type schema to every constant (from $\mathcal{C}$) and every variable (from $\mathcal{V}$) of the λ-calculus.

We start with an initial type assumption A_0:

$A_0(\text{not}) = \text{Bool} \to \text{Bool}$

$A_0(+) = \text{Int} \to \text{Int} \to \text{Int}$

$A_0(5) = \text{Int}$

$A_0(x) = \forall a.\ a$ (for any $x \in \mathcal{V}$)

We disregard typeclasses here and assume that there are different $+$-functions for Int, Float, ...

For data List a = Nil | Cons a (List a):

$A_0(\text{Nil}) = \forall a.\ \text{List}\ a$

$A_0(\text{Cons}) = \forall a.\ a \to (\text{List}\ a) \to (\text{List}\ a)$

Def 4.1.1 (Type Schemas and Type Assumptions)

Type schemas are built according to the following grammar:

$$
\begin{aligned}
\underline{\text{typeschema}} \rightarrow\ & (\underline{\text{tyconstr}}\ \underline{\text{typeschema}}_1 \ldots \underline{\text{typeschema}}_n),\ n \geq 0 \\
\mid\ & (\underline{\text{typeschema}}_1 \rightarrow \underline{\text{typeschema}}_2) \\
\mid\ & (\underline{\text{typeschema}}_1, \ldots, \underline{\text{typeschema}}_n),\ n \geq 0 \\
\mid\ & \underline{\text{var}} \\
\mid\ & \forall \underline{\text{var}}.\ \underline{\text{typeschema}}
\end{aligned}
$$

For a typeschema τ with free variables $a_1, \ldots, a_n$, we write $\forall \tau$ for $\forall a_1 \ldots \forall a_n.\ \tau$.

A type assumption A is a (possibly partial) function from $\mathcal{V} \cup \mathcal{C}$ to the set of type schemas.
A type assumption A with $A(x_i) = \tau_i$ for $1 \leq i \leq n$ which is undefined on other arguments is also written $A = \{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$.

The initial type assumption A_0 is defined on Slide 59.

Here, we assume that $\underline{\text{Constr}}$ is a user-defined data constructor introduced by:

data $\underline{\text{tyconstr}}$ $a_1 \dots a_m$ = ... | $\underline{\text{Constr}}$ $\underline{\text{type}}_1 \dots \underline{\text{type}}_n$ | ...

For two type assumptions A and A', we define $A + A'$ as: $(A+A')(x) = \begin{cases} A'(x), & \text{if } A'(x) \text{ is defined} \\ A(x), & \text{otherwise} \end{cases}$