We now present an algorithm \( W \) for type checking under a certain type assumption.

\[ W(A, t) \]

\( \uparrow \)

\( \uparrow \)

type assumption \( \lambda \)-term

checks whether \( t \) is well typed under the type assumption \( A \). If yes, then \( W \) returns the result \((\Theta, t)\).

\( \Theta \) most general type of \( t \) under the type assumption \( \Theta(A) \).

We also write \( A\Theta \)

We now introduce the algorithm \( W \) for the different forms of \( \lambda \)-terms \( t \). (Slide 60)

4.2.1. Type Inference for Variables and Constants

Now \( t \) is \( c \in \text{Cuv} \).

If the type assumption contains \( c :: \text{Var}, \ldots, \text{Var} \ldots \text{c} \) where \( \text{c} \) is a type (without "\( \text{Var} \)"), then the most general
type of \( c \) is \( \tau \). Here, we rename the type variables \( a_1, ..., a_n \) to fresh variables \( b_1, ..., b_n \) that do not occur free in \( A \) or \( \tau \).

To make \( c \) well typed, the current type assumption \( A \) does not need to be refined, i.e., \( \Theta \) is the identity \( \text{id} \).

\[
\begin{align*}
    \nu(A_0, x) &= (\text{id}, b) \quad \text{for } x \in \nu \quad A_0(x) = \forall a.e \\
    \nu(A_0, \text{not}) &= (\text{id}, \text{Bool} \to \text{Bool}) \\
    \nu(A_0, \text{Cons}) &= (\text{id}, c \to (\text{list } c) \to (\text{list } c)) \\
    A_0(\text{Cons}) &= \\
    &= \forall a. a \to (\text{list } a) \to (\text{list } a) \\
\end{align*}
\]

**General Rule for \( c \in \text{C U V} \):**

\[
\nu(A + \{c::\forall a_1, ..., a_n. \tau\}, c) = (\text{id}, \tau \exists a_1/b_1, ..., a_n/b_n) 
\]

where \( b_1, ..., b_n \) are fresh variables

4.2.2 Type Inference for Lambda Abstractions

**Idea:** To determine the type of \( \lambda x.t \):

- assume that we know the type of \( x \) (i.e., \( x \) has some type \( b \))
- extend the previous type assumption by \( x::b \)
- under this assumption, compute the type \( \tau \) of \( t \)

Moreover, we compute a subst \( \Theta \) which states how our type assumption...
Then the whole term has type $b\theta \rightarrow \tau$.

**General Rule for $\lambda x.t$:**

$$\sigma(A, \lambda x.t) = (\theta, b\theta \rightarrow \tau),$$

where $\sigma(A + \{x::b\}, t) = (\theta, \tau)$,

$b$ is a fresh type variable.

We now illustrate this rule (and type checking for $\lambda$-abstractions) with several examples.

**Ex:**

\[
\begin{array}{c}
\lambda f. \text{plus} (f \text{ True}) (f \text{ 3}) \\
\uparrow \quad \uparrow \quad \uparrow \\
\text{type Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{type Bool} \\
\text{type Int}
\end{array}
\]

So $f$ must have the type schema $\forall a. a \rightarrow \text{Int}$.

Thus, the whole term has the type schema:

$$(\forall a. a \rightarrow \text{Int}) \rightarrow \text{Int} \quad \text{this is the type of functions to Int}$$

Where the argument is a function of type $a \rightarrow \text{Int}$.

What functions have the type schema $\forall a. a \rightarrow \text{Int}$?

$\lambda x. 1$, but not $\lambda x. x$, $\lambda x. x + 1$, ...
So there is an important difference between the following type schemas:

\((\forall a. a \rightarrow \text{Int}) \rightarrow \text{Int}\)

\(\forall a. (a \rightarrow \text{Int}) \rightarrow \text{Int} \iff \text{If \ a \ fct. \ has \ this \ type, then \ it \ also \ has \ the \ types} \)

\((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}, (\text{Bool} \rightarrow \text{Int}) \rightarrow \text{Int}, \ldots\)

E.g.: \(\lambda f. \ \text{plus} \ (f \ \text{bot}) \ (f \ \text{bot})\)

Type a

In Haskell, one is restricted to **shallow** type schemas.

**Def 42.1 (Shallow Type Schemas)**

A type schema is shallow iff it has the form

\(\forall a_1, \ldots, a_n. T\) where \(T\) is a type (i.e., \(T\) does not contain quantifiers).

Most prog. languages are restricted to shallow type schemas to make type checking decidable.

\(\Rightarrow \lambda f. \ \text{plus} \ (f \ \text{True}) \ (f \ \text{3})\)

is not well typed in Haskell. (i.e., \(\lambda x.t\))
Consequence: A variable bound by \( \lambda \) must have the same type for all (free) occurrences of \( x \) in \( t \).

\[ \lambda f. \text{plus} \ (f \ True) \ (f \ 3) \] is not well typed

Let \( f = \lambda x \to 1 \) in \( \text{plus} \ (f \ True) \ (f \ 3) \) is well typed

We now execute \( \text{let} \) for some examples:

**Ex:** \( \lambda x. \text{Cons} \ 0 \ (\text{Cons} \ x \ x) \)

\( \text{type} \ e \to \text{list} e \to \text{List} e \)

is not well typed: \( x \) must get the same type everywhere in \( \text{Cons} \ 0 \ (\text{Cons} \ x \ x) \)

To compute \( \text{let} \ (A_0, \lambda x. \text{Cons} \ 0 \ (\text{Cons} \ x \ x)) \), we need to compute \( \text{let} \ (A_0 + \{x :: b\}, \text{Cons} \ 0 \ (\text{Cons} \ x \ x)) \) which fails.

Note that we have \( x :: b \), not \( x :: A_6.b \)

Indeed, \( \lambda x. \text{Cons} \ 0 \ (\text{Cons} \ x \ x) \) has the (non-shallow) type schema: \( (A_6.b) \to (\text{List} \ A_7) \)

**Ex:** \( \text{let} \ (A_0, \lambda x. \text{tuple}_2 \ x \ x) = (\text{id}, \text{b \to} (b, b)) \), since
\[ w(A_0 + \{x \mapsto b\}, \text{tuple}_2 \times x) = (\text{id}, (b, b)) \]

\[ \text{Ex: } w(A_0, \lambda x. \text{plus} \times x) = (\frac{b}{\text{Int}}, \frac{\text{Int}}{\text{Int}}) \]

\[ w(A_0 + \{x \mapsto b\}, \text{plus} \times x) = (\frac{b}{\text{Int}}, \frac{\text{Int}}{\text{Int}}) \]

\[ A_0 \text{ (plus): Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

The current type assumption must be refined (by instantiating \( b \) with \( \text{Int} \)) to make \( \text{plus} \times x \) well-typed.

### 4.2.3 Type Inference for Applications

\((t_1 \ t_2)\): If \( t_1 \) has type \( \tau_1 \), and \( t_2 \) has type \( \tau_2 \), then check whether \( \tau_1 \) corresponds to \( \tau_2 \rightarrow \tau_3 \).

Then the result has type \( \tau_3 \).

\[ \text{Ex: (not True): } \text{Here, "not" has type } \frac{\tau_1}{\text{Bool} \rightarrow \text{Bool}} \]

\[ \text{and "True" has type } \frac{\tau_2}{\text{Bool}}. \]

Thus, \( \tau_1 \) corresponds to \( \tau_2 \rightarrow \tau_3 \) for \( \tau_3 = \text{Bool}. \) So term is well-typed.
and has type \( \tau_3 = \text{Bool} \).

In general, "correspondence" of \( \tau_1 \) and \( \tau_2 \rightarrow \tau_3 \) might involve instantiation of type variables.

Ex: \textbf{Cons 0}

\textit{Cons} has type \( \tau_1 = e \rightarrow \text{list} e \rightarrow \text{list} e \)

0 has type \( \tau_2 = \text{Int} \)

Does \( \tau_1 \) 'correspond' to \( \tau_2 \rightarrow \tau_3 \) for some type \( \tau_3 \)?

This means:

Is there a substitution \( \Theta \) of type variables by types such that \( \tau_1 \Theta = (\tau_2 \rightarrow b) \Theta \) holds for a fresh type variable \( b \)?

This means: We search for a \underline{unifier} \( \Theta \) of \( \tau_1 \) and \( \tau_2 \rightarrow b \).

In the example \textbf{Cons 0}:

\( \tau_1 : e \rightarrow \text{list} e \rightarrow \text{list} e \)

\( \tau_2 : \text{Int} \)

Unifier of \( \tau_1 \) and \( \tau_2 \rightarrow b \) is \( \frac{\left[ e/\text{Int} \right]}{0} \)

Resulting type of \textbf{Cons 0} is \( b \Theta : \text{list} \text{Int} \rightarrow \text{list} \text{Int} \).

\underline{Def 4.22 (Unification)}
Let \( \Theta \) be a substitution of type variables by types. The subst. \( \Theta \) is a unifier of two types \( \tau_1 \) and \( \tau_2 \) if
\[
\tau_1 \Theta = \tau_2 \Theta.
\]
A subst. \( \Theta' \) is most general unifier (mgu) of \( \tau_1 \) and \( \tau_2 \) iff
- \( \Theta' \) is a unifier of \( \tau_1 \) and \( \tau_2 \)
- for all unifiers \( \Theta \) of \( \tau_1 \) and \( \tau_2 \), there exists a subst. \( \Gamma \) with \( \Theta = \Theta' \Gamma \).

First apply the mgu \( \Theta' \), then apply a more special subst. \( \Gamma \).

We write \( \Theta' = \text{mgu}(\tau_1, \tau_2) \).

Ex: Cons bot

We have to unify \( \tau_1 = \text{e} \rightarrow \text{liste} \rightarrow \text{liste} \)
with \( \tau_2 = \text{a} \rightarrow \text{b} \)

Possible unifiers: \( \Theta_1 = [a/e, b/\text{liste} \rightarrow \text{liste}] \).

Resulting type of Cons bot is \( b \Theta_1 = \text{liste} \rightarrow \text{liste} \)

Which leads to the most general type

\( \Theta_2 = [a/\text{Int}, e/\text{Int}, b/\text{listInt} \rightarrow \text{listInt}] \).

Resulting type of Cons bot is \( b \Theta_2 = \text{list Int} \rightarrow \text{list Int} \).

General Rule for type-checking applications \((\tau_1, \tau_2)\):
\[ \omega(A, (t_1, t_2)) = (\Theta_1, \Theta_2, \Theta_3, \delta \Theta_3), \]

where \( \omega(A, t_n) = (\Theta_n, \tau_n) \)

\[ \omega(A \Theta_n, t_2) = (\Theta_2, \tau_2) \]

\[ \Theta_3 = \text{mgu}(\tau_n, \Theta_2, \tau_2 \rightarrow \delta) \]

\( b \) is a fresh variable

c to make \( t_1 \) well-typed, one has to refine \( A \) to \( A \Theta_1 \). This should be taken into account when type-checking \( t_2 \)

to make \( t_2 \) well-typed, the type assumption must be refined further by \( \Theta_2 \).
Then \( t_1 \) has the type \( \tau_n \Theta_2 \).

Ex: \( \lambda x. \text{Cons} \times x \)

\[ \omega(A_0, \lambda x. \text{Cons} \times x) = \]

\[ \omega(A_0 + \{x \mapsto b\}, (\text{Cons} \times x)) = \]

\[ \omega(A_0 + \{x \mapsto b\}, \text{Cons} \times x) = ([\text{Id} e, b'/\text{Liste} \rightarrow \text{Liste}] \text{Liste} \rightarrow \text{Liste}) \]

\[ \omega(A_0 + \{x \mapsto b\}, \text{Cons}) = (\text{Id} e, \text{Liste} \rightarrow \text{Liste} \rightarrow \text{Liste}) \]

\[ \omega(A_0 + \{x \mapsto b\}, x) = (\text{Id} e, b) \]

\[ \text{mgu}(e \rightarrow \text{Liste} \rightarrow \text{Liste}, b \rightarrow b') = [b/e, b'/\text{Liste} \rightarrow \text{Liste}] \]

\[ \omega((A_0 + \{x \mapsto b\})[b/e, b'/\ldots]), x) = (\text{Id} e, b) \]

\[ A_0 + \{x \mapsto e\} \]

\[ \text{mgu}(\text{Liste} \rightarrow \text{Liste}, e \rightarrow b'') \] fails ("Occur failure")

\[ \Rightarrow \text{term is not well typed!} \]

4.2.4 The Full Type Inference Algorithm
The algorithm \( \mathcal{W} \) is a modified version of the \( \mathcal{W} \)-algorithm by R. Milner (1978).

**Theorem 4.23** (Correctness of \( \mathcal{W} \))

Let \( t \) be a \( \lambda \)-term over the constants \( \mathcal{C} \) from Def 3.3.4 where \( \mathcal{W}(A_0, t) = (\Theta, \tau) \).

Let \( t \xrightarrow{\betaSS} t' \) where \( \Delta \) is the Delta-Rule-Set for Haskell from Def 3.3.5. Then \( \mathcal{W}(A_0, t') = (\Theta', \tau') \) for some \( \Theta' \).

This means: Static type checking ensures that no type errors are introduced at runtime.