Type checking in the lambda cube

In the lectures, we presented a function $\mathsf{type}_\Gamma(M)$ that 'computes' the type in the calculus of constructions of a term M in a context Γ:

$$\begin{aligned}
\mathsf{type}_\Gamma(*) &= \Box \\
\mathsf{type}_\Gamma(x) &= \Gamma(x) \\
\mathsf{type}_\Gamma(\lambda x : A.\, M) &= \Pi x : A.\, \mathsf{type}_{\Gamma,x:A}(M) \\
\mathsf{type}_\Gamma(\Pi x : A.\, B) &= \mathsf{type}_{\Gamma,x:A}(B) \\
\mathsf{type}_\Gamma(A \to B) &= \mathsf{type}_\Gamma(B) \\
\mathsf{type}_\Gamma(FM) &= B[x := M] \qquad \text{if } \mathsf{type}_\Gamma(F) =_\beta \Pi x : \mathsf{type}_\Gamma(M).\, B
\end{aligned}$$

However, this function does *not* establish whether its argument is well-typed. These equations are only meaningful when both the context Γ and the term M are well-typed already.

Also, this function is partial. The notation $\Gamma(x)$ stands for the type A of the last occurrence of the form $x : A$ in the context Γ, and if the variable x does not occur in Γ, the expression $\Gamma(x)$ is undefined.

We present here a variant of this function that also type-checks its arguments. We define *two* functions, that both are *total*: a function $\mathsf{ok}(\Gamma)$ which returns a Boolean, and the function $\mathsf{type}_\Gamma(M)$ which returns either a term, or the constant $\mathsf{Failure}$. In these functions, now Γ can be an arbitrary precontext and M an arbitrary preterm:

$$\begin{aligned}
\mathsf{type}_\Gamma(*) &= \Box && \text{if } \mathsf{ok}(\Gamma) \\
\mathsf{type}_\Gamma(x) &= \Gamma(x) && \text{if } \mathsf{ok}(\Gamma) \text{ and } x : A \in \Gamma \\
\mathsf{type}_\Gamma(\lambda x : A.\, M) &= \Pi x : A.\, \mathsf{type}_{\Gamma,x:A}(M) && \text{if } \mathsf{type}_{\Gamma,x:A}(M) = B \text{ and} \\
& && \mathsf{type}_\Gamma(\Pi x : A.\, B) \in \{*, \Box\} \\
\mathsf{type}_\Gamma(\Pi x : A.\, B) &= \mathsf{type}_{\Gamma,x:A}(B) && \text{if } \mathsf{type}_\Gamma A = s_1 \text{ and} \\
& && \mathsf{type}_{\Gamma,x:A}(B) = s_2 \text{ and} \\
& && (s_1, s_2, s_2) \in \mathcal{R} \\
\mathsf{type}_\Gamma(A \to B) &= \mathsf{type}_\Gamma(B) && \text{if } \mathsf{type}_\Gamma A = s_1 \text{ and} \\
& && \mathsf{type}_\Gamma(B) = s_2 \text{ and} \\
& && (s_1, s_2, s_2) \in \mathcal{R}
\end{aligned}$$

$$\mathsf{type}_\Gamma(FM) = B[x := M] \qquad \text{if } \mathsf{type}_\Gamma(F) \twoheadrightarrow_\beta \Pi x : A.\, B \text{ and } \mathsf{type}_\Gamma M =_\beta A$$

$$\mathsf{type}_\Gamma(M) = \mathsf{Failure} \qquad \text{when none of these cases applies}$$

$$\mathsf{ok}(\cdot) = \mathsf{true}$$

$$\mathsf{ok}(\Gamma,\, x : A) = \mathsf{type}_\Gamma(A) \in \{*, \Box\}$$

This definition refers to a set $\mathcal{R}$ of *rules*, which depends on the type theory. Here is a table of the rules $\mathcal{R}$ for the type theories from Femke's course notes:

type theory	*set of rules*
$\lambda\!\!\rightarrow$	$\mathcal{R} = \{(*,*,*)\}$
$\lambda\mathsf{P}$	$\mathcal{R} = \{(*,*,*), (*,\Box,\Box)\}$
$\lambda 2$	$\mathcal{R} = \{(*,*,*), (\Box,*,*)\}$

Finally, here are some properties of these functions, that we give without proof:

Proposition (Soundness)**.**

$$\mathsf{type}_\Gamma(M) = A \quad \Longrightarrow \quad \Gamma \vdash M : A$$

Proposition (Completeness)**.**

$$\Gamma \vdash M : A \quad \Longrightarrow \quad \mathsf{type}_\Gamma(M) =_\beta A$$

Proposition.

$$\mathsf{type}_\Gamma(M) = \mathsf{Failure} \quad \Longrightarrow \quad M \textit{ is not typable in } \Gamma$$

Proposition. *The function* $\mathsf{type}_\Gamma(M)$ *terminates, even when* Γ *and* M *are not well-typed.*

This last proposition considers the mutually recursive definitions of $\mathsf{type}_\Gamma(M)$ and $\mathsf{ok}(\Gamma)$ as an algorithm. There is a subtlety here: the terms A and B in the application case $\mathsf{type}_\Gamma(FM)$ are not uniquely determined. However, one can define a terminating reduction algorithm that computes specific A and B (if they exist) and fails otherwise, which can be used to make this specific.