\[
M ::= c \mid x \mid M \mid \lambda x : \tau.M
\]

where \(x\) are variables and \(c\) integer or boolean constants. An expression of this language is called QST expression.

In [12], Thatte presents a rule based type inference system and an algorithm that he calls \textit{Type}, which he claims types the same terms as the rule based system. Here we show both as inference rules, and show that contrary to his claim, they do not type the same terms.

We now present the typing system, called here QST, which has two phases. The first phase transforms an expression given in the object language into an expression of an “internal” language. This internal language consists of the object language together with coercions, which we define now:

**Definition 1** For each pair of types \(\tau_1 \) and \(\tau_2\) such that \(\tau_1 \leq \tau_2\) there are coercions \(\gamma_{\tau_1}^{\tau_2}\) and \(\delta_{\tau_2}^{\tau_1}\).

These coercions are inserted where there is the possibility of a type mismatch during run time, according to the typing rules given below. In each typing rule the notation \(\text{T}(e) \vdash \eta : \tau\) reads “under the assumptions \(\text{T}\) the object expression \(e\) can be transformed into the internal expression \(\eta\) with type \(\tau\).”

\[
\text{T}(e) \vdash \eta : \tau \\
\text{T}(e) \vdash \eta : \tau \\
\text{T}(e) \vdash \eta : \tau \\
\text{T}(e) \vdash \eta : \tau
\]

The second phase of the QST typing system tries to type expressions that could lead to run time type error. This phase, called plausibility checking, can be described as a confluent terminating set of rewrite rules. These rules are shown below. The object wrong means run time type error, and it is the only term that cannot be assigned a type.

\[
e \eta_{=e}^{\tau} : e \\
e \eta_{\tau_{1}}^{\tau} : e \\
\]

The expression \(\tau_1 \cap \tau_2\) denotes the greatest lower bound of types \(\tau_1\) and \(\tau_2\).

To illustrate how QST works, we now present two examples:

**Example 1** \(\lambda x : \Omega. x x\)

As an example of how this system works consider the expression \(\lambda x : \Omega. x x\). The typing of this expression is:

\[
x \in \text{Dom}(\lambda x : \Omega) \\
x \in \text{Dom}(\lambda x : \Omega) \\
x \in \text{Dom}(\lambda x : \Omega) \\
x \in \text{Dom}(\lambda x : \Omega) \\
\text{T}(\lambda x : \Omega. x x) \vdash \lambda x : \Omega. x x : \Omega \\
\]

If we rewrite the expression in the following way, \(\lambda x : \Omega \rightarrow \Omega. x x\), then the resulting internal expression is \(\lambda x : \Omega \rightarrow \Omega. x x : \Omega^{\tau_{1}}_{\tau_{2}}\).

**Example 2** \((\lambda x : \Omega \rightarrow \Omega. x x)(\lambda x : \Omega. x x)\)

The resulting internal expression is: \((\lambda x : \Omega \rightarrow \Omega. (x x : \tau_{1}^{\tau_{2}}_{\tau_{1}}))(\lambda x : \Omega. (x x : \tau_{1}^{\tau_{2}}_{\tau_{1}}))\).

In [12], Thatte gives an algorithm called \textit{Type} that integrates in one pass the typing rules and the plausibility checking phase. Given an expression in the object language, this algorithm returns an expression in the internal language and a type for it. Here we show a set of inference rules that are equivalent to the algorithm \textit{Type}, and we call this system in the same way.

\[
\text{T}(e) \vdash \eta : \tau \\
\text{T}(e) \vdash \eta : \tau \\
\text{T}(e) \vdash \eta : \tau
\]

If \(M\) is an expression, \(\text{T}\) is a type environment and \(\tau\) is a type, then we write \(\text{T}(\eta) \vdash \tau : \tau\)

to say that \(\tau\) is the type assigned to \(M\), which is converted to \(M\) by the above typing rules.

Consider now the following expression:

\[(\lambda x : \text{int} + \text{aux} x) \text{ true}\]