6.6. POLYMORPHIC TYPE SYSTEMS AND TYPED $\mu$SCHEME

Type checking

This book does not provide a type checker for Typed $\mu$Scheme; its implementation is left as Exercise 14. Type checking requires an expression or definition, a type environment, and a kind environment. Calling $\texttt{elabdef}(t, \Gamma, \Delta)$ should return a pair $(\Gamma', s)$, where $(t, \Gamma) \rightarrow \Gamma'$ and $s$ is a string that represents the type of the thing defined.

\begin{verbatim}
(type checking for Typed \mu Scheme \[prototype\] \[283\a\] =

exception LeftAsExercise of string
fun typeof _ = raise LeftAsExercise "typeof"
fun elabdef _ = raise LeftAsExercise "elabdef"
\end{verbatim}

6.6.7 The rest of an interpreter for Typed $\mu$Scheme

Evaluation

Here is an appropriate place to dispose of the evaluation rules for Typed $\mu$Scheme. The rules for expressions are exactly the same as the rules for $\mu$Scheme; at run time, the types have no effect whatever. We require new rules for type abstraction and application, but the evaluator behaves exactly as if these constructs aren’t there.

\begin{align*}
\langle e, \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle & \text{(TYAPPLY)} \\
\langle \text{TYP}\text{APPLY}(e, \tau_1, \ldots, \tau_n), \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle \\
\langle e, \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle & \text{(TYLAMBDA)} \\
\langle \text{TY}\text{LAB}\text{MA}\text{N}\text{DA}(\tau_1, \ldots, \tau_n, e), \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle
\end{align*}

Most of the evaluator for Typed $\mu$Scheme is just like the evaluator for $\mu$Scheme in Chapter 5. The code for the two new cases acts as if TYAPPLY and TYLAMBDA aren’t there.

\begin{verbatim}
(alternatives for \texttt{ev} for TYAPPLY and TYLAMBDA \[283\b\] \[715\d\] =
| \texttt{ev} (TYAPPLY (_, _)) = \texttt{ev} e \\
| \texttt{ev} (TYLAMBDA (_, e)) = \texttt{ev} e
\end{verbatim}

The rest of the evaluator appears in Appendix C.

The rules for definitions are slightly different from those in $\mu$Scheme; as described in Exercise 38 in Chapter 3, VAL must always create a new binding. Otherwise, we could subvert the type system by using VAL to change the type of an existing value. The VAL rule must use the old environment; VAL-Rec uses the new one.

\begin{align*}
\ell \not\in \text{dom} \sigma \\
\langle e, \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle & \text{(VAL)} \\
\langle \text{VAL}(x, e), \rho, \sigma \rangle & \rightarrow \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\} \rangle \\
\ell \not\in \text{dom} \sigma \\
\langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\} \rangle & \Downarrow \langle v, \sigma' \rangle & \text{(VAL-Rec)} \\
\langle \text{VAL-REC}(x, \tau, e), \rho, \sigma \rangle & \rightarrow \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\} \rangle
\end{align*}

The code that implements these rules is in Appendix C.