We have the same new handlers as in Typed Impcore.

(more read-eval-print handlers)

| TypeError msg => continue ("type error: " ^ msg)
| BugInTypeChecking msg => continue ("bug in type checking: " ^ msg)

 Initializing the interpreter

To put everything together into a working interpreter, we need an initial kind environment as well as a type environment and a value environment.

(initialization for Typed µScheme)

val initialEnvs =
  let fun addPrim ((name, prim, funty), (types, values)) =
    ( bind (name, funty, types),
      bind (name, ref (PRIMITIVE prim), values) )
  val (types, values) = foldl addPrim (emptyEnv, emptyEnv)
  fun addVal ((name, v, ty), (types, values)) =
    ( bind (name, ty, types),
      bind (name, ref v, values) )
  val (types, values) = foldl addVal (types, values)

  fun addKind ((name, kind), kinds) = bind (name, kind, kinds)
  val kinds = foldl addKind emptyEnv
  fun addEnv ((name, env), (types, values)) =
    ( bind (name, env, types),
      bind (name, ref env, values) )
  val envs = (kinds, types, values)

  val basis = (ML representation of initial basis (automatically generated))
  val reader = defreader (false, stringsreader ("initial basis", basis), tuschemeSyntax)
  in readCheckEvalPrint (reader, fn _ => (), fn _ => ()) envs end

The code for the primitives appears in Appendix G. It is similar to the code in Chapter 5 except that it supplies a type, not just a value, for each primitive.

The function runInterpreter takes one argument, which tells it whether to prompt.

(initialization for Typed µScheme)

fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
    fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
    val reader = defreader (noisy, filereader ("standard input", TextIO.stdIn), tuschemeSyntax)
  in
    ignore (readCheckEvalPrint (reader, writeln, errorln) initialEnvs)
  end
6.10.3 Learning about polymorphic type systems

13. Write a type checker for Typed μScheme. That is, implement \texttt{elabdef} in code chunk \texttt{268a}. Although you could write this checker by cloning and modifying the type checker for Typed Impcore, you may have better luck building a checker from scratch by following the type rules for Typed μScheme.

When you type-check \texttt{val-rec}(x, τ, e), be sure to check that \(x\) is not evaluated in \(e\), as described on page \texttt{267}.

When you type-check literals, use the rules on page \texttt{265}. Although these rules are incomplete, they should suffice for anything the parser can produce. If a literal \texttt{PRIMITIVE} or \texttt{CLOSURE} reaches your type checker, the impossible has happened, and your code should raise an appropriate exception.

14. Suppose we get sick and tired of writing \@ signs everywhere, so we decide to extend Typed μScheme by making \texttt{PAIR}, \texttt{FST}, and \texttt{SND} abstract syntax instead of functions.

(a) What is the type of the following function?

\[
\begin{align*}
\texttt{(type-lambda ('a) (type-lambda ('b) (lambda (((pair a b) p)) (pair (snd p) (fst p))))))}
\end{align*}
\]

(b) Using the type rules from the chapter, give a derivation tree proving the correctness of your answer to part \texttt{14a}.

15. A great advantage of a polymorphic type system is that the language can be extended without touching the abstract syntax, the values, the type checker, or the evaluator. Without changing any of these parts of Typed μScheme, extend Typed μScheme with a \texttt{queue} type constructor and the polymorphic values \texttt{empty-queue}, \texttt{empty?}, \texttt{put}, \texttt{get-first}, and \texttt{get-rest}. (A more typical functional queue provides a single \texttt{get} operation which returns a pair containing both the first element in the queue and any remaining elements. If instead you write the two functions \texttt{get-first} and \texttt{get-rest}, you won’t have to fool with pair types.)

(a) What is the kind of the type constructor \texttt{queue}? Add it to the initial \(\Delta\) for Typed μScheme.

(b) What are the types of \texttt{empty-queue}, \texttt{empty?}, \texttt{put}, \texttt{get-first}, and \texttt{get-rest}?

(c) Add them to the initial \(\Gamma\) and \(\rho\) of Typed μScheme. You will need to write implementations in ML.

16. Without changing the abstract syntax, values, type checker, or evaluator of Typed μScheme, extend Typed μScheme with the \texttt{pair} type constructor and the polymorphic functions \texttt{pair}, \texttt{fst}, and \texttt{snd}.

(a) What is the kind of the type constructor \texttt{pair}? Add it to the initial \(\Delta\) for Typed μScheme.

(b) What are the types of \texttt{pair}, \texttt{fst}, and \texttt{snd}?

(c) Add them to the initial \(\Gamma\) and \(\rho\) of Typed μScheme. As you add them to \(\rho\), you can use the same implementations that we use for \texttt{cons}, \texttt{car}, and \texttt{cdr}.