Implementing Exceptions
Kathi Fisler, WPI
October 6, 2003

1 Implementing Exceptions

Consider an example of using exceptions to terminate a product computation if one argument evaluates to 0. In Scheme we might write this as follows:

(define (prod L)
  (cond [(empty? L) 1]
        [(cons? L) (cond [(eq? 0 (first L)) (raise 0)]
                        [else (* (first L) (prod (rest L))))]]))

(define (real-prod L)
  (with-handlers ([lambda (exn) (number? exn)]
                  [lambda (exn) (printf "\text{\textquotedblright}\text{a}\text{\textquotedblright}exn 0])
        (prod L)))))

Scheme’s \texttt{with-handlers} is fairly general because it allows a program to test whether a handler applies to a given exception. Let’s implement a similar version that has the handler, but not the test for whether to use a handler. In particular, we will introduce two new language constructs: \texttt{raise} to throw exceptions and \texttt{try} to specify where to catch exceptions. If we also introduce list operators \texttt{kons}, \texttt{kar}, and \texttt{kdr} corresponding to \texttt{cons}, \texttt{first}, and \texttt{rest}, we could write the \texttt{prod} example in our concrete syntax as follows:

{\texttt{rec}} {\texttt{prod}} {\texttt{fun}} L
  {\texttt{ifempty}} L {1}
    {\texttt{if0}} {\texttt{kar}} L
      {\texttt{raise}} 0
        {\texttt{*}} {\texttt{kar}} L {\texttt{prod}} {\texttt{kdr}} L
  {\texttt{with}} {\texttt{real-prod}} {\texttt{fun}} L {\texttt{try}} {\texttt{prod}} L
    {\texttt{fun}} exn exn
  {\texttt{real-prod}} {\texttt{kons}} 4 {\texttt{kons}} 0 {\texttt{kons}} 5 {\texttt{mt-list}})

How do we go about supporting this example in our language? Obviously, we need to add \texttt{raise} and \texttt{try} to the abstract syntax, parser, and interpreter. How does the interpreter handle these? Consider \texttt{raise} – it needs to “return” the raised value while indicating that the value is not a normal return value. To handle this, we’ll introduce a new kind of value into our language, called \texttt{exnV}.

(\texttt{define-datatype}} FWA-value FWA-value?
  [numV (n number?)]
  [closureV (param symbol?)
          (body FWAE?)
          (cache SubCache?)]
  [exnV (v FWA-value?)])

*\textit{drawing on notes from sk/dbtucker, Brown CS}
Now, every time we call interp, we must check whether the returned value is an \textit{exnV} or a regular return value. If it's an \textit{exnV}, we want to ignore the context and return it. Otherwise, we continue the computation as before. For example, the \textit{add} case would now look like:

\[
\text{add (lhs rhs)}
\hspace{1em}
(\text{let ([lv (interp lhs sc)]})
\hspace{1em}
(\text{cases FWA-value lv}
\hspace{1em}
[\text{exnV (v) lv}]
\hspace{1em}
[\text{else (let ([rv (interp rhs sc)]})
\hspace{1em}
(\text{cases FWA-value rv}
\hspace{1em}
[\text{exnV (v) rv}]
\hspace{1em}
[\text{else (numV+ lv rv)])}]))}
\]

In the \textit{try} case, if the value of \textit{interp} on the expression to try is an exception, invoke the associated handler on that expression. Otherwise, just return the value normally.