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 "~a" exn) 0)])
    (prod L)))
```

Scheme's **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: *raise* to throw exceptions and *try* to specify where to catch exceptions. If we also introduce list operators *kons*,*kar*, and *kdr* corresponding to *cons*, *first*, and *rest*, we could write the *prod* example in our concrete syntax as follows:

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

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

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

* drawing on notes from sk/dbtucker, Brown CS

Now, every time we call interp, we must check whether the returned value is an *exnV* or a regular return value. If it's an *exnV*, we want to ignore the context and return it. Otherwise, we continue the computation as before. For example, the *add* case would now look like:

```
[add (lhs rhs)
    (let ([lv (interp lhs sc)])
       (cases FWA-value lv
         [exnV (v) lv]
         [else (let ([rv (interp rhs sc)])
                 (cases FWA-value rv
                   [exnV (v) rv]
                   [else (numV+ lv rv)]))]))]
```

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