Semantics of Exceptions

Administrative notes:

- Homework 4 is available online.
- Project: send a 1 paragraph abstract of your project and a pointer to your project home page to me (yelick@cs).

1. Termination Semantics: the handler receives control from the signaller and keeps it. Used in ML, Common Lisp, and CLU.
2. Resumption Semantics: the handler received control from the signaller, but later returns it to the signaller. Used in OS traps.

FLK! Exceptions

First, a simple exception handling mechanism with resumption semantics.

$$\begin{array}{rl} E ::= \ldots & |\ (\texttt{raise}\ I_{except}\ E_{val}) \\ & |\ (\texttt{trap}\ I_{except}\ E_{handler}\ E_{body}) \\ & |\ (\texttt{error}\ Y_{symbol}) \end{array}$$

- ($\texttt{raise}$ I_{except} E_{val}) raises an exception named I_{except} with value E_{val}.
- ($\texttt{trap}$ I_{except} $E_{handler}$ E_{body}) evaluates E_{body}. If an exception named I_{except} is raised during this evaluation, the value of the raise expression is the result of applying the handler procedure, $E_{handler}$, to the value passed during the raise.
- ($\texttt{error}$ Y_{symbol}) terminates the program and returns the given symbol as the answer.

Exception Handling Example

```
(define add
  (lambda (x y)
    (let ((check-integer
            (lambda (a)
             (if (integer? a) a (raise non-integer a)))))
      (+ (check-integer x) (check-integer y)))))

(define sum-list
  (lambda (lst)
    (if (null? lst)
        0
        (add (car lst) (sum-list (cdr list))))))


(sum-list '(1 2 3))
```


Example Continued

```
(define simple-handler
  (lambda (x)
    (cond ((bool? x) (if x 1 0)
          ((symbol? x) 10)
          (else (error not-bool-or-sym)))))

(trap non-integer simpler-handler
   (sum-list '(5 yes #t)))

(trap non-integer simpler-handler
   (sum-list '(5 ())))
```

Scoping of Exceptions

The intended semantics of `trap` are based on dynamic scoping – it should pick the innermost definition of `a` in the first example.

```
(trap a (lambda (x) (+ 4000 x))
  (trap b (lambda (x) (+ 300 (raise a (+ x 4))))
    (trap a (lambda (x) (+ 20 x))
       (+ 1 (raise b 2)))))  ---> 327

(trap a (lamda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4)))) --> 421
```

We define a new semantic domain to represent the exception handler namespace. Do we need a separate environment?

$$
\begin{aligned}
&w \in \mathit{Handler\text{-}Env} = \mathit{Identifier} \rightarrow \mathit{Procedure}\\
&p \in \mathit{Procedure} = \mathit{Denotable} \rightarrow \mathit{Computation}\\
&c \in \mathit{Computation} = \mathit{Handler\text{-}End} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}
\end{aligned}
$$

Semantics of Exceptions

The valuation functions for the exception mechanism are:

$$
\begin{aligned}
&\mathcal{E}[\![(\texttt{trap}\ I_{except}\ E_{handler}\ E_{body})]\!]\\
&\quad = \lambda e\ w\ k\ .\ (\mathcal{E}[\![E_{handler}]\!]\ e\ w\ (\mathit{test\text{-}procedure}\\
&\qquad\qquad (\lambda p\ .\ (\mathcal{E}[\![E_{body}]\!]\ e\\
&\qquad\qquad\quad (\mathit{extend\text{-}handlers}\ w\ I_{except}\ p)\ k))))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![(\texttt{raise}\ I_{except}\ E_{handler}\ E_{body})]\!]\\
&\quad = \lambda ewk\ .\ (\mathcal{E}[\![E_{body}]\!]\ e\ w\\
&\qquad\qquad (\lambda v\ .\ ((\mathit{get\text{-}handler}\ w\ I_{except})\ v\ w\ k)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![(\texttt{error}\ Y_{message})]\!]\\
&\quad = \lambda e\ w\ k\ .\ (\mathit{error\text{-}cont}\ \mathcal{Y}[\![Y_{message}]\!])
\end{aligned}
$$

What would you change to make the exception names statically scoped? What would you change for termination semantics?

The Call/cc or cwcc Construct

`cwcc` can be simulated using the non-local jump construct from last time:

```
(define (cwcc proc)
   (label here
      (proc (lambda (val) (jump here val)))))

(define (prod-list a-list)
  (cwcc
    (lambda (return)
       (letrec ((prod (lambda (lst)
                     (cond ((null? lst) 1)
                           ((= 0 (car lst)) (return 0))
                           (else (* (car lst)
                                    (prod (cdr lst)))))))
          (prod a-list)))))
```

- `(prod-list '(2 3 4))`
- `(prod-list '(2 0 yow)`
- `(prod-list '(2 yow 0)`

Continuation Passing Style

Continuation passing style is also used as a compilation target:

- Makes control explicit.
- Allows for unusual control flow, e.g., multi-threading.
- Exposes tail recursion.
- Used in compiling higher order languages.

Implementing Recursion

```
(define fact (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

Evaluate `(fact 4)`:

```
(fact 4)
=> (* 4 (fact 3))
=> (* 4 (* 3 (fact 2)))
=> (* 4 (* 3 (* 2 (fact 1))))
=> (* 4 (* 3 (* 2 (* 1 (fact 0)))))
=> (* 4 (* 3 (* 2 (* 1 1))))
=> 24
```

Increasing space is needed at each step in the evaluation, corresponding to the call frames on the program stack.

Tail Recursion

In iterative, or tail-recursive form:

```
(define fact-iter (lambda (n a)
   (if (zero? n) a (fact-iter (- n 1) (* n a)))))
```

The implementation can use a constant amount of space by reusing the same stack fram on recursive calls:

```
(fact-iter 4 1)
=> (fact-iter 3 4)
=> (fact-iter 2 12)
=> (fact-iter 1 24)
=> (fact-iter 0 24)
=> 24
```

`fact-iter` is tail recursive, because the last operation performed by the function is a recursive call.

It has iterative control behavior.

Implementing Recursion

We can further modify the program to eliminat the arguments in calls.

```
(define fact-iter
  (lambda (n) (let ((a 1))
    (letrec ((loop (lambda ()
       (if (zero? n) a (begin (set! a (* n a))
                              (set! n (- n 1))
                              (loop))))))
      (loop)))))
```

In this version, recursive calls are simply "goto's."

CPS Conversion

We can define a small language that enforces continuation passing style (CPS). The `A` expressions are values and the `C` are general expressions.

```
A ::= I | L | (lambda (I) C) | (prim P I*)
C ::= A | (C A) | (if I C1 C2) \\
```

Cannot nested applications in this language:

- `((lambda (x) x) 3)` is OK.
- `((lambda (x) x) ((lambda (y) y) 3))` is not.

In CPS, each function take a continuation as an "extra" argument.

CPS Conversion

Consider a simplified FLK:

```
E ::= I | L | (lambda I E) | (primop P E) |
      (if E0 E1 E2) | (E1 E2)
```

Define a function `Cvt` to convert to CPS language.

```
Cvt [I] = (lambda (k_return) (k_return I))

Cvt [L] = (lambda (k_return) (k_return L))

Cvt [(primop P E)] = (lambda (k_return)
                        (Cvt[E] (lambda (v)
                          (k_return (prim P v)))))

Cvt [(primop P E1 E2)] = (lambda (k_return)
                          (Cvt[E1] (lambda (v1)
                            (Cvt[E2] (lambda (v2)
                              (k_return
                               (prim P E1 E2)))))))
```


Conversion to CPS

```
Cvt [(if E0 E1 E2)] = (lambda (k_return)
             (Cvt [E0] (lambda (b)
                   (if b (Cvt[E1] k_return)
                         (Cvt[E2] k_return)))))


Cvt [(lambda (I) E)] = (lambda (k_return)
             (k_return (lambda (k_call I)
                         (Cvt [E] k_call))))


Cvt [(E1 E2)] = (lambda (k_return)
                  (C[E1] (lambda (fn)
                     (Cvt[E2] (lambda (v)
                                (fn k_return v))))))
```