# Today's Lecture Notes for cs173

sk and dbtucker

October 4, 2000

## Continuation passing style

Let's revisit our function which computes the product of a list of numbers:

```
(define (Π L)
  (cond
    [(empty? L) 1]
    [(cons? L) (cond
                 [(zero? (first L)) 0]
                 [else (* (first L) (Π (rest L)))])]))
```

Using this version, (Π '(1 2 3 0 4 5 6)) still has to do 3 multiplications. In our previous solution, we eliminated these multiplications by using *let/cc*, which created an escaper that could be invoked when Π comes across a 0.

Is it possible to eliminate the multiplications without using *let/cc*? Remember what *let/cc* did—it allowed us to change the context in which we were evaluating the current expression. If we explicitly pass around a context, can we mimic the behavior of *let/cc* and the escaper? Let's give it a shot. We'll add a third parameter $k$ to our function Π which represents the context:

```
(define (Π L k)
  ...)
```

First, what do we do when $L$ is empty? In this case, the product is 1, and we need to hand this value to the context (i.e. the remaining computation):

```
(define (Π L k)
  (cond
    [(empty? L) (k 1)]
    [(cons? L) ...]))
```

If $L$ is not empty, we look at (*first* $L$). If it is 0, then we know the whole product is 0. We could hand the value 0 to the context, but then we would still perform the extraneous multiplications like in the first example. The whole point of passing the context $k$ is that we want to *ignore* it and just return 0 when we encounter a 0. So we have:

```
(define (Π L k)
```

```
  (cond
    [(empty? L) (k 1)]
    [(cons? L) (cond
                 [(zero? (first L)) 0]
                 [...])]))
```

Now we're almost done. In the recursive call to Π, we need to extend the context. The context should take the result of the computation (■), and multiply it by (*first* $L$):

```
(define (Π L k)
  (cond
    [(empty? L) (k 1)]
    [(cons? L) (cond
                 [(zero? (first L)) 0]
                 [else (Π (rest L) (λ (■)
                                     (* (first L)
                                        ■))))])]))
```

Let's try to evaluate (Π '(1 2 3) $I$), where $I$ is just the identity function, (λ (■) ■):

```
(Π '(1 2 3) I)
⇒ (Π '(2 3) (λ (■) (* 1 ■)))
⇒ (Π '(3) (λ (■) (* 2 ■)))
⇒ (Π '() (λ (■) (* 3 ■)))
⇒ ((λ (■) (* 3 ■)) 1)
⇒ 3
```

Oops! The new context we create (i.e., (λ (■) ...)) needs to hand its result (i.e., (* (*first* $L$) ■)) to the context that was passed in ($k$). Here's the final (correct) version:

```
(define (Π L k)
  (cond
    [(empty? L) (k 1)]
    [(cons? L) (cond
                 [(zero? (first L)) 0]
                 [else (Π (rest L) (λ (■)
                                     (k (* (first L)
                                           ■))))])]))
```

**Exercise:** Verify (by writing out the contexts) that (Π '(1 2 3) $I$) does indeed compute to 6.

The function Π is written in *continuation passing style*, or *CPS*, since it explicitly passes a continuation (context).

## A continuation-passing interpreter

Let's write our interpreter in continuation passing style. We'll first tackle the easy cases—when the result is a value, just hand it to the context.

```
(define (interp a d k)
  (cases AFunExp a
    [numE (n) (k (numV n))]
    [varE (v) (k (get-sub v d))]
    [addE (le re) ...]
    [funE (param body) (k (funV param body d))]
    [appE (fe ae) ...]))
```

What do we do in the *addE* case? We want to evaluate *le* in some context...

```
(interp le d ...)
```

...which takes that result (call it *lv*) and evaluates *re* in another context...

```
(interp le d
        (λ (lv)
          (interp re d ...)))
```

...which takes the result of evaluating *re* (call it *rv*) and adds the two values...

```
(interp le d
        (λ (lv)
          (interp re d
                  (λ (rv)
                    (numV+ lv rv)))))
```

...and then hands the result to the original context (*k*)...

```
(interp le d
        (λ (lv)
          (interp re d
                  (λ (rv)
                    (k (numV+ lv rv))))))
```

One important note: we now have control over the order of evaluation. Previously, the order of evaluation in our interpreter depended on Scheme's order of evaluation. Now our interpreter explicitly computes *le* first, then computes *re*, so the order is left-to-right.

Here's what we have so far:

```
(define (interp a d k)
  (cases AFunExp a
    [numE (n) (k (numV n))]
    [varE (v) (k (get-sub v d))]
    [addE (le re) (interp le d
                          (λ (lv)
                            (interp re d
```

```
                                        (λ (rv)
                                          (k (numV+ lv rv))))))]
    [funE (param body) (k (funV param body d))]
    [appE (fe ae) ...]))
```

Now we consider the *appE* case. Perhaps it is just like the *addE* case?

```
(interp fe d
        (λ (fv)
          (interp ae d
                  (λ (av)
                    (k (apply-fun fv av))))))
```

Not quite. Remember that *apply-fun* calls *interp* on the body of the function, so that is where we should send the context:

```
(interp fe d
        (λ (fv)
          (interp ae d
                  (λ (av)
                    (apply-fun fv av k))))))
```

where *apply-fun* is now:

```
(define (apply-fun fv av k)
  (cases AFunVal fv
    [funV (param body env)
          (interp body (new-sub param av env) k)]))
```

Here's the final version of the CPS interpreter:

```
(define (interp a d k)
  (cases AFunExp a
    [numE (n) (k (numV n))]
    [varE (v) (k (get-sub v d))]
    [addE (le re) (interp le d
                          (λ (lv)
                            (interp re d
                                    (λ (rv)
                                      (k (numV+ lv rv))))))]
    [funE (param body) (k (funV param body d))]
    [appE (fe ae) (interp fe d
                          (λ (fv)
                            (interp ae d
                                    (λ (av)
                                      (apply-fun fv av k)))))]))
```

**Exercise:** How would you add *let/cc* to this interpreter?

## Implementing exceptions

Let's add exceptions to our CPS interpreter. Recall that the datatype is:

```
(define-datatype AFunExp AFunExp?
  [numE (n number?)]
  [varE (v symbol?)]
  [addE (lhs AFunExp?)
        (rhs AFunExp?)]
  [funE (param symbol?)
        (body AFunExp?)]
  [appE (fun AFunExp?)
        (arg AFunExp?)]
  [tryE (body AFunExp?) (handler AFunExp?)]
  [raiseE (val AFunExp?)])
```

We also need an exception value:

```
(define-datatype AFunVal AFunVal?
  [numV (n number?)]
  [funV (param symbol?)
        (body AFunExp?)
        (env DSub?)]
  [exnV (x AFunVal?)])
```

We know how to interpret the expressions that evaluate to values. Next, consider the *raiseE* case. We want to evaluate the argument expression, then create an exception with that value. The exception is then passed to the continuation:

```
[raiseE (val) (interp val d
                      (λ (xv)
                        (k (exnV xv))))]
```

This isn't quite right. If the body of *raiseE* itself raises an exception, we should just hand that exception to the continuation:

```
[raiseE (val) (interp val d
                      (λ (xv)
                        (cond
                          [(exn-value? xv) (k xv)]
                          [else (k (exnV xv))]))))]))
```

Now let's do the *tryE* case. First, the body is evaluated. If the result is not an exception, we just pass it to the continuation. If the result is an exception, we evaluate the handler and apply it to the value contained in the exception:

```
[tryE (body handler) (interp body d
                             (λ (bv)
                               (cases AFunVal bv
                                 [exnV (xv) (interp handler d
                                                    (λ (hv)
```

```
                                                    (apply-fun hv xv k)))]
                                        [else (k bv)])))]
```

Note, however, that the evaluation of the handler could itself be an exception. So we need to check for this situation:

```
[tryE (body handler) (interp body d
                          (λ (bv)
                            (cases AFunVal bv
                              [exnV (xv) (interp handler d
                                                 (λ (hv)
                                                   (cond
                                                     [(exn-value? hv) (k hv)]
                                                     [else (apply-fun hv xv k)])))]
                              [else (k bv)])))]
```

All that's left is to rewrite the cases such as *addE*, where we need to check whether the subexpressions evaluate to exceptions. This can be a bit tedious:

```
[addE (le re) (interp le d
                  (λ (lv)
                    (cond
                      [(exn-value? lv) (k lv)]
                      [else (interp re d
                                    (λ (rv)
                                      (cond
                                        [(exn-value? rv) (k rv)]
                                        [else (k (numV+ lv rv))]))))]))))]
```

The *appE* case is similar. Our final interpreter looks like this:

```
(define (interp a d k)
  (cases AFunExp a
    [numE (n) (k (numV n))]
    [varE (v) (k (get-sub v d))]
    [addE (le re) (interp le d
                          (λ (lv)
                            (cond
                              [(exn-value? lv) (k lv)]
                              [else (interp re d
                                            (λ (rv)
                                              (cond
                                                [(exn-value? rv) (k rv)]
                                                [else (k (numV+ lv rv))]))))]))))]
    [funE (param body) (k (funV param body d))]
    [appE (fe ae) (interp fe d
                          (λ (fv)
                            (cond
                              [(exn-value? fv) (k fv)]
```

```
                    [else (interp ae d
                                (λ (av)
                                  (cond
                                    [(exn-value? av) (k av)]
                                    [else (apply-fun fv av k)]))))]))))]
[tryE (body handler) (interp body d
                           (λ (bv)
                             (cases AFunVal bv
                               [exnV (xv) (interp handler d
                                                (λ (hv)
                                                  (cond
                                                    [(exn-value? hv) (k hv)]
                                                    [else (apply-fun hv xv k)]))))]
                               [else (k bv)]))))]
[raiseE (val) (interp val d
                  (λ (xv)
                    (cond
                      [(exn-value? xv) (k xv)]
                      [else (k (exnV xv))]))))]))
```