Programming Abstractions

Lecture 33: Continuation Passing Style 2

Stephen Checkoway

CPS guidelines for recursive procedures

Continuations are procedures with 1 argument

The recursive procedure has a continuation parameter, `k`

The continuation argument is called once for each branch of computation (think base case and recursive case)

- Not calling the continuation on one of the cases is a common mistake

At the top-level, the continuation is usually identity

Recursive calls must be tail-recursive

Reverse in CPS

```
(define (reverse-k lst k)
  (cond [(empty? lst) (k empty)]
        [else (reverse-k (rest lst)
                         (λ (x) (k (append x (list (first lst))))))]))
```

Note: this is spectacularly inefficient

- `(reverse lst)` takes time O(n) where n is the length of the list
- `(reverse-k lst identity)` takes time $O(n^2)$

Append in CPS

```
(define (append-k lst1 lst2 k)
  (cond [(empty? lst1) (k lst2)]
        [else (append-k (rest lst1)
                        lst2
                        (λ (x) (k (cons (first lst1) x))))]))
```

What is the run time of `append-k`?

```
(define (append-k lst1 lst2 k)
  (cond [(empty? lst1) (k lst2)]
        [else (append-k (rest lst1)
                        lst2
                        (λ (x) (k (cons (first lst1) x))))]))
```

Let m be the length of `lst1` and n be the length of `lst2`

A. O(1)

B. O(m)

C. O(n)

D. O($m + n$)

E. O(mn)

Comparing append in CPS to normal recursion

```
(define (append-k lst1 lst2 k)
  (cond [(empty? lst1) (k lst2)]
        [else (append-k (rest lst1)
                        lst2
                        (λ (x) (k (cons (first lst1) x))))]))
(define (append lst1 lst2)
  (cond [(empty? lst1) lst2]
        [else (cons (first lst1)
                    (append (rest lst1) lst2))]))
```

In `append`, the continuation of the recursive call is `(cons (first lst1) □)` *plus* all of the other earlier recursive calls (example on next slide)

This is identical to the passed-in continuation in `append-k` where `k` is going to perform the work of the other recursive calls

Continuation example

Appending '(1 2 3) to '(a b c)

Step	`lst1`	append's recursive continuation	k argument to append-k's recursive call (expanded)
0	`'(1 2 3)`	`(cons 1 □)`	`(λ (x) (k (cons 1 x)))`
1	`'(2 3)`	`(cons 1 (cons 2 □))`	`(λ (x) (k (cons 1 (cons 2 x))))`
2	`'(3)`	`(cons 1 (cons 2 (cons 3 □)`	`(λ (x) (k (cons 1 (cons 2 (cons 3 x)))))`
3	`'()`	—	—

- append's continuations also include the top-level continuation the table omits
- `k` in append-k's recursive calls aren't expanded, they're the closure `(λ (x) (k (cons (first lst1) x)))` with `k` bound to the previous closure and `lst1` bound to the corresponding `lst1` argument in the table
- CPS makes the continuations explicit

Let's write some CPS

`(map-k f lst k)`

Implement the `map-k` function using CPS

Let's think about types

- `lst` : list of α
- `f` : $\alpha \rightarrow \beta$
- For recursive calls `k` : list of $\beta \rightarrow$ list of β
- For the top-level `k` : list of $\beta \rightarrow \gamma$

Hints:

- The continuation you pass to the recursive call to map-k takes as its argument the result of making the recursive call
- Cons `(f (first lst))` onto the this result and pass that as an argument to `k`

So what good is this?

Programming with explicit continuations gives you a lot of control

- E.g., you can *ignore* the continuation that is built up and do something else!

Consider our standard sum procedure

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))
```

Suppose we want to modify this to return `#f` if `lst` contains an element that isn't a number

What goes wrong with this approach?

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [(not (number? (first lst))) #f]
        [else (+ (first lst) (sum (rest lst)))]))
```

A. Nothing. It's perfect

B. `(sum '(foo 1 2 3))` will fail

C. `(sum '(1 2 foo 3))` will fail

D. B and C

A working attempt with CPS

Since CPS uses tail-recursion, we can ignore our built-up continuation k and just return #f

Normal base case uses (k •)

Error case does not call k

```
(define (sum-k lst k)
  (cond [(empty? lst) (k 0)]
        [(not (number? (first lst))) #f]
        [else (sum-k (rest lst)
                     (λ (x) (k (+ x (first lst)))))]))

(sum-k '(1 2 3 foo 4) identity) => #f
```

A better approach

We can use an error continuation

- This lets the caller decide what to do with the error

```
(define (sum-k lst k err)
  (cond [(empty? lst) (k 0)]
        [(not (number? (first lst))) (err (first lst))]
        [else (sum-k (rest lst)
                     (λ (x) (k (+ x (first lst))))
                     err)]))
```

Normal base case uses (k •)

Error case uses (err •)

```
> (sum-k '(1 2 3 foo 4)
         identity
         (λ (bad) (printf "Bad element: ~s\n" bad)))
Bad element: foo
```

CPS is similar to callbacks in languages like JavaScript

Example

```
const promise = new Promise((resolve, reject) => {
    // perform some computation
    if (computation_was_successful) {
        resolve(success_value);
    } else {
        reject(failure-value);
    }
});
```

`resolve` is the continuation for a successful computation
`reject` is the error continuation

Write some more CPS

`(collatz-k n k)`: CPS version of `collatz`

- Two recursive cases to handle, must call k in both

`(fib-k n k)`: CPS version of `fib`

- Implement the (very slow) recursive version but using CPS
- Tricky because we need to make two recursive calls
- Continuation for the first recursive call should make the second recursive call
- Continuation for the second recursive call should add the results of both recursive calls together and pass that to `k`

Write more CPS!

`(map/error f lst k err)`

In this case, the user-supplied `(f x k err)` takes three arguments:

- x: an element of the list
- k: the continuation f should call on success
- err: the continuation f should call on error

When map/error calls f, it must pass it a continuation that will make a recursive call to map/error with the rest of the list

The continuation for the recursive call to map/error must combine the results of the calls to f and map/error into a list

```
(map/error (λ (x k err) (if (zero? x) (err x) (k (add1 x))))
           '(1 2 0 3 4)
           identity
           (λ (bad-element) #f)) => #f
```