Programming Abstractions

Lecture 35: Call With Current Continuation

Stephen Checkoway

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`

From earlier

A continuation is determined by the expression's evaluation context at run time

```
(define (fact n)
  (cond [(zero? n) 1]
        [else (* n (fact (sub1 n)))]))
```

At the point 1 is evaluated in the call `(fact 0)`, the continuation is □

At the point 1 is evaluated in the call `(fact 1)`, the continuation is `(* 1 □)`

At the point 1 is evaluated in the call `(fact 2)`, the continuation is `(* 2 (* 1 □))`

Key: The continuation is **all** the rest of computation

The current continuation

At every point in a computation the **current continuation** is the continuation of whatever expression is currently being evaluated

The current continuation is constantly changing

Example

```
(define (fact n)
  (cond [(zero? n) 1]
        [else (* n (fact (sub1 n)))]))
(fact 3)
```

redex	current continuation	value
(fact 3)	□	—
(zero? 3)	(cond [□ 1][else (* 3 (fact (sub1 3)))])	#f
(* 3 (fact (sub1 3)))	□	—
(fact (sub1 3))	(* 3 □)	—

Example: continued

redex	**current continuation**	**value**
(fact 3)	□	—
(zero? 3)	(cond [□ 1][else (* 3 (fact (sub1 3)))])	#f
(* 3 (fact (sub1 3)))	□	—
(fact (sub1 3))	(* 3 □)	—
(sub1 3)	(* 3 (fact □))	2
(fact 2)	(* 3 □)	—
(zero? 2)	(* 3 (cons [□ 1][else (* 2 (fact (sub1 2)))])	#f
(* 2 (fact (sub1 2)))	(* 3 □)	—
(fact (sub1 2))	(* 3 (* 2 □))	—

Example: continued

`redex`	current continuation	value
`(fact (sub1 2))`	`(* 3 (* 2 □))`	—
`(sub1 2)`	`(* 3 (* 2 (fact □)))`	1
`(fact 1)`	`(* 3 (* 2 □))`	—
`(zero? 1)`	`(* 3 (* 2 (cons [□ 1][else (* 1 (fact (sub1 1)))])))`	#f
`(* 1 (fact (sub1 1)))`	`(* 3 (* 2 □))`	—
`(fact (sub1 1))`	`(* 3 (* 2 (* 1 □)))`	—
`(sub1 1)`	`(* 3 (* 2 (* 1 (fact □))))`	0
`(fact 0)`	`(* 3 (* 2 (* 1 □)))`	—
`(zero? 0)`	`(* 3 (* 2 (* 1 (cons [□ 1][else (* 0 (fact (sub1 0)))]))))`	#t

Example: continued

`redex`	current continuation	value
`(zero? 0)`	`(* 3 (* 2 (* 1 (cons [□ 1][else (* 0 (fact (sub1 0)))]))))`	#t
`1`	`(* 3 (* 2 (* 1 □)))`	1
`(* 1 1)`	`(* 3 (* 2 □))`	1
`(* 2 1)`	`(* 3 □)`	2
`(* 3 2)`	`□`	6

Example: simplified

Let's just look at the recursive calls

`redex`	current continuation	value
`(fact 3)`	□	—
`(fact 2)`	`(* 3 □)`	—
`(fact 1)`	`(* 3 (* 2 □))`	—
`(fact 0)`	`(* 3 (* 2 (* 1 □)))`	1
`(* 1 1)`	`(* 3 (* 2 □))`	1
`(* 2 1)`	`(* 3 □)`	2
`(* 3 2)`	□	6

Example 2: With an accumulator

```
(define (fact-a n acc)
  (cond [(zero? n) acc]
        [else (fact-a (sub1 n) (* n acc))]))
(fact-a 3 1)
```

`redex`	current continuation	value
`(fact-a 3 1)`	□	—
`(fact-a 2 3)`	□	—
`(fact-a 1 6)`	□	—
`(fact-a 0 6)`	□	6

Tail-recursive calls

In the first example, the current continuation changes at each recursive call

In the second example, the current continuation doesn't change at the recursive calls

- It does fluctuate a bit as sub-expressions like `(* n acc)` are evaluated

Current continuation of general recursion grows with each recursive call

Current continuation of tail-recursion remains constant with each recursive call

call-with-current-continuation
call/cc

Call with current continuation

Scheme gives the programmer programatic access to the current continuation

```
(call-with-current-continuation proc)
(call/cc proc)
```

- `proc` is a 1-argument procedure
- `proc` is called with the current continuation as an argument

Call/cc

`(call/cc (λ (k) body))`

When this is evaluated

- it calls the λ with the current continuation as the argument
- within `body`, calling `k` with a value, `(k value)`, immediately returns from `call/cc` with `value` as the result
- if `k` is not called in `body`, the return from `call/cc` has the value of `body`

Examples

```
(call/cc (λ (k) (k 42)))
```

`k` is called with value 42 => result is 42

```
(call/cc (λ (k) 10))
```

`k` is not called, so the result just the body, namely 10

Less simple example

```
(call/cc (λ (k) (* 5 3 (k 2))))
```

`k` is called with the value 2, so the result *is* 2

What is the value of this expression?

```
(+ 1 (call/cc (λ (k)
                ((λ (x) (* 20 (k x)))
                 3))))
```

A. 3

B. 4

C. 60

D. 61

E. 81

Escaping from recursion

Remember our example summing elements of a list

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

(sum-cc '(1 2 3 4)) => 10
(sum-cc '(1 2 steve 4)) => #f
```

Revisiting index-of with a fold

```
(define (index-of x lst)
  (call/cc (λ (k)
             (foldl (λ (y idx)
                      (if (equal? x y)
                          (k idx) ; Return idx from call/cc
                          (add1 idx)))
                    0
                    lst)
             -1))) ; Return -1 from call/cc

(index-of 4 '(0 1 4 2 3 4 5)) ; returns 2
```

We can store the current continuation

```
(define exit-k 0)
(call/cc (λ (k) (set! exit-k k)))

(define (prod-cc lst)
  (cond [(empty? lst) 1]
        [(not (number? (first lst))) (exit-k #f)]
        [else (* (first lst) (prod-cc (rest lst)))]))

(prod-cc '(1 2 3 4 #t 6)) ; returns #f
```

Continuations are deeply weird

```
(define A 0)
(set! A (call/cc identity))
(define B A)
```

This defines A and B to be the continuation `(set! A □)`

If I call `(A 10)`, it runs that continuation, setting `A` to be 10

If I call `(B 25)`, it runs the continuation again, setting `A` to be 25

There is so much more to this

```
(call-with-composable-continuation proc)
```

```
(dynamic-wind pre-thunk value-thunk post-thunk)
```

prompts

aborts

…