ITP20005
Implementing Continuations

Lecture26
JC

Summary from your classmates and others

- Continuation
  - Rest of computation to be evaluated from one point.
  - Rest of work that has to happen to finish the evaluation of a program
  - Abstract representation of the control state of a program.
- Continuation Passing Style (CPS)
  - Easy to transform your representation of stacks from the actual stack to heap
    - So, if you have a deep recursion, you wouldn't be run out of stack memory
  - Can simulate control flow like operators, exceptions, loops,...

Summary from your classmates and others

- call/cc
  - Call with current continuation
  - Racket's way to deal with continuation
  - Must be passed a procedure 'p' of one argument. It constructs a concrete representation of the current continuation and passes it to p. The continuation itself is represented by a procedure k.
- let/cc
  - Simplified call/cc syntax
    - (call/cc (lambda (k) (k 2))
    - (let/cc k (k 2))

call/cc example

- call with current continuation

```
(define retry #f)

(define factorial
        (lambda (x)
              (if (= x 0)
                  (call/cc (lambda (k) (set! retry k) 1))
                  (* x (factorial (- x 1))))))


(factorial 4) ; Result -> 24
(retry 1) ; Result -> 24
(retry 2) ; Result -> 48
```

let/cc example

- call with current continuation

```
(define retry #f)

(define factorial
        (lambda (x)
                (if (= x 0)
                    (let/cc k (set! retry k) 1)
                    (* x (factorial (- x 1))))))


(factorial 4) ; Result -> 24
(retry 1) ; Result -> 24
(retry 2) ; Result -> 48
```

call/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10)
(+ (* (call/cc
        (lambda (k)
          (k 2))) 3) 10)

(+ (* (call/cc
        (lambda (k)
          (set! retry k) 2)) 3) 10)

(retry 3) ; =>
(retry 2)
(retry 1)
```

call/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10 )   ;; ⇒ 16
(+ (* (call/cc
                    (lambda (k)
                            (k 2))) 3) 10)   ;; ⇒ 16

(+ (* (call/cc
                    (lambda (k)
                            (set! retry k) 2)) 3) 10) ;; ⇒ 16

(retry 3) ;; => 19
(retry 2) ;; ⇒ 16
(retry 1) ;; ⇒ 13
```

let/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10 )                    ;; 16
(+ (* (let/cc k (k 2)) 3) 10)      ;; 16

(+ (* (let/cc k (set! retry k) 2) 3) 10)    ;; 16

(retry 3) ;; =>  19
(retry 2) ;; => 16
(retry 1) ;; => 13
```

let/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10 )                   ;; 16
(+ (* (let/cc k (k 2)) 3) 10)     ;; 16

(+ (* (let/cc k (set! retry k) 2) 3) 10)    ;; 16

(retry 3) ;; =>  19
(retry 2) ;; => 16
(retry 1) ;; => 13
```

We are implementing a language that supports let/cc operator!!
We start this implementation from FAE!
⇒ **KCFAE**

call/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10 )                         ;; 16
(+ (* (let/cc k (k 2)) 3) 10)      ;; 16

(+ (* (let/cc k (set! retry k) 2) 3) 10)    ;; 16

(retry 3) ;; =>  19
(retry 2) ;; => 16
(retry 1) ;; => 13
```

KCFAE Grammar

```
<KCFAE> ::= <num>
          | {+ <KCFAE> <KCFAE>}
          | {- <KCFAE> <KCFAE>}
          | <id>
          | {fun {<id>} <KCFAE>}
          | {if0 <KCFAE> <KCFAE> <KCFAE>}
          | {withcc <id> <KCFAE>}
```

FAE + Continuations (K) + Conditional expression (C)

KCFAE Grammar

```
<KCFAE> ::= <num>
          | {+ <KCFAE> <KCFAE>}
          | {- <KCFAE> <KCFAE>}
          | <id>
          | {fun {<id>} <KCFAE>}
          | {if0 <KCFAE> <KCFAE> <KCFAE>}
          | {withcc <id> <KCFAE>}
```

```
{withcc k {+ 1 {k 2}}}
```

KCFAE Grammar

<KCFAE> ::= <num>
| {+ <KCFAE> <KCFAE>}
| {- <KCFAE> <KCFAE>}
| <id>
| {fun {<id>} <KCFAE>}
| {if0 <KCFAE> <KCFAE> <KCFAE>}
| {withcc <id> <KCFAE>}

{withcc k {+ 1 {k 2}}} ⇒ 2

KCFAE Values

```
(define-type KCFAE-Value
  [numV       (n number?)]
  [closureV  (param symbol?)
             (body KCFAE?)
             (ds DefrdSub?)]
  [contV     (c procedure?)])
```

Implementing withcc

```
; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp kcfae ds)
   (type-case KCFAE kcfae
      ...
      [withcc (id body-expr)
                    ...]
       ...))
```

Implementing withcc

```
; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp kcfae ds)
  (type-case KCFAE kcfae
    ...
    [withcc (id body-expr)
            (...
            (interp body-expr
                    (aSub id
                          ...
                          ds)))]
    ...))
```

Implementing withcc

```
; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp kcfae ds)
  (type-case KCFAE kcfae
    ...
    [withcc (id body-expr)
            (...
            (interp body-expr
                    (aSub id
                          (contV ...)
                          ds)))]
    ...))
```

Implementing withcc

```
; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp kcfae ds)
  (type-case KCFAE kcfae
    ...
    [withcc (id body-expr)
            (let/cc k
            (interp body-expr
                    (aSub id
                          (contV k)
                          ds)))]
     ...))
```

Implementing withcc

```
; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp kcfae ds)
  (type-case KCFAE kcfae
    ...
    [withcc (id body-expr)
            (let/cc k
            (interp body-expr
                    (aSub id
                          (contV k)
                          ds)))]
    ...))
```

This will work, but it's too meta-circular to tell us anything.

Implementing KCFAE from scratch

- Steps to implementation
  - Making continuations explicitly in the interpreter.
    - Need to change our interpreter based on CPS
      - so that we can access to the continuation at every stage.
      - Interpreter takes an extra argument k (continuation) a.k.a a receiver.
        - We want 'interp' to communicate its answer by passing it to the given k.
  - Providing access to continuations in an extended language. ⇒ withcc

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    [num (n)     ... (numV n)...]
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    [num (n)     (k (numV n))]
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    [num (n)     (k (numV n))]
    ...
    [id  (s)     (k (lookup s ds))]
    [fun (p b)   (k (closureV (lambda (a-val dyn-k)
                         (interp b (aSub p a-val ds) dyn-k))))]
```

A function can be called in different contexts. Since we should get that context dynamically from the surrounding context, we get its continuation as dyn-k.

KCFAE Values

```
(define-type KCFAE-Value
  [numV       (n number?)]
  [closureV  (p procedure?)]
  [contV     (c procedure?)])
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [add (l r)  … (num+ (interp l ds… ) (interp r ds… ) … ]
    …
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [add (l r)  (k (num+ (interp l ds… ) (interp r ds… ))]
    …
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [add (l r)  (k (num+ (interp l ds… ) (interp r ds… ))]
    …
```

We can't call interp in the midst of some larger computation for continuations.

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [add (l r)  (interp l ds
                       (lambda (lv)
                         (k (num+ lv (interp r ds …  )))))]
    …
```

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [add (l r)  (interp l ds
                        (lambda (lv)
                          (interp r ds
                            (lambda (rv)
                              (k (num+ lv rv))))))]
    …
```

So, (1) there must not be an interp call in the sub-expression position. (2) interp for rhs must have the third parameter.

In this way, we can wrap the entire execution context for constitutions.

Implementing KCFAE

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [sub (l r)  (interp l ds
                  (lambda (lv)
                    (interp r ds
                      (lambda (rv)
                        (k (num- lv rv))))))]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [if0 (test t f) … ]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [if0 (test t f) (interp test ds
                       ...
                         (interp t ds ...)
                         (interp f ds ...)]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [if0 (test t f) (interp test ds
                     (lambda (tv)
                       (if(eq? (interp test ds … ) (numV 0))
                         (interp t ds ...)
                         (interp f ds ...))))]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
    [if0 (test t f) (interp test ds
                   (lambda (tv)
                      (if(eq? (interp test ds k) (numV 0))
                         (interp t ds k)
                         (interp f ds k))))]
    …
```

Implementing KCFAE

```
; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)
(define (interp fae ds k)
  (type-case KCFAE fae
    …
   [app (f a)   … ]
    …
```

Implementing KCFAE

```
; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)
(define (interp fae ds k)
  (type-case KCFAE fae
    …
   [app (f a)   (interp f ds
                   … (interp f ds …)
                   … (interp a ds …)]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
   [app (f a)   (interp f ds
                    (lambda (f-val)
                     (interp a ds
                         (lambda (a-val)
                            ...))))]
    …
```

Implementing KCFAE

```
; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)
(define (interp fae ds k)
  (type-case KCFAE fae
    …
   [app (f a)   (interp f ds
                  (lambda (f-val)
                    (interp a ds
                       (lambda (a-val)
                         (type-case KCFAE-Value f-val
                           [closureV (c) (c a-val … )]
                           [contV (c) (c a-val)]
                           [else (error … )]))))))]
    …
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    ...
   [app (f a)   (interp f ds
                       (lambda (f-val)
                         (interp a ds
                                 (lambda (a-val)
                                   (type-case KCFAE-Value f-val
                                     [closureV (c) (c a-val k)]
                                     [contV (c) (c a-val)]
                                     [else (error "not an applicable value")])))))]
    ...
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
     [withcc (cont-var body)
                    … ]
    ))
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
     [withcc (cont-var body)
                 … (interp body
                       (aSub cont-var
                            (contV … )
                            ds)
                       … )]
  ))
```

Implementing KCFAE

```
; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)
(define (interp fae ds k)
  (type-case KCFAE fae
    …
     [withcc (cont-var body)
                (interp body
                        (aSub cont-var
                              (contV (lambda (val)
                                        (k val)))
                              ds)
                        … )]
   ))
```

Implementing KCFAE

; interp: KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha

; or interp: KCFAE DefrdSub receiver -> doesn't return (but receiver returns)

```
(define (interp fae ds k)
  (type-case KCFAE fae
    …
     [withcc (cont-var body)
               (interp body
                     (aSub cont-var
                          (contV (lambda (val)
                                 (k val)))
                          ds)
                     k)]
   ))
```

Calling interp with a continuation

- How do we start calling interp with a continuation?

Calling interp with a continuation

- How do we start calling **interp** with a continuation?

  (interp kcfae (mtSub) lambda (x) x))

```
(define (run sexp ds)
    (interp (parse sexp) ds (lambda (x) x))

(run '{withcc k {+ 1 {k 3}}} (mtSub))
```

KCFAE Grammar

```
<KCFAE> ::= <num>
          | {+ <KCFAE> <KCFAE>}
          | {- <KCFAE> <KCFAE>}
          | <id>
          | {fun {<id>} <KCFAE>}
          | {if0 <KCFAE> <KCFAE> <KCFAE>}
          | {withcc <id> <KCFAE>}
```

```
{withcc done                          ;; done = {fun {x}  x}
    {{withcc esc                      ;; esc = {fun {y} {y 3}}
        {done {+ 1 {withcc k          ;; k = {fun {z} {{done {+ 1 z}} 3}}
                      {esc k}}}}}}
      3}}
```

call/cc (easier example)

```
#lang racket
(define retry #f)

(+ (* 2 3) 10 )                         ;; 16
(+ (* (let/cc k (k 2)) 3) 10)           ;; 16

(+ (* (let/cc k (set! retry k) 2) 3) 10)    ;; 16

(retry 3) ;; =>  19
(retry 2) ;; => 16
(retry 1) ;; => 13
```

let/cc
⇒ withcc in our language, KCFAE

Topics we cover and schedule (tentative)

- Racket tutorials (L2,3, HW)
- Modeling languages (L4,5, HW
- Interpreting arithmetic (L5)
- Language principles
  - Substitution (L6, HW)
  - Function (L7)
  - Deferring Substitution (L8,L9)
  - First-class Functions (L10-12)
  - Laziness (L13, L14)
  - Recursion (L15, L16)
  - Mutable data structures (L17,18,19,20)
  - Variables (L21, L22)
  - Continuations (L23~L26)
- Guest Video Lecture

No class: October 2 (Fri, Chuseok), October 9 (Fri, Hangul day)
Online only class can be provided.

TODO

Read Chapter 23. Semantics

JC

jcnam@handong.edu

https://lifove.github.io

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or created by JC based on the main text book.