Chapter 5. Continuation-Passing Interpreters (Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Outline

The concept of environment in Ch.3 establishes the data context of program execution, and the concept of *continuation* here does the control context of program execution.

- ♢ A Continuation-Passing Interpreters
- ♢ A Trampolined Interpreter
- ♢ An Imperative Interpreter
- ♢ Exceptions
- ♢ Threads

Introduction: Data Context

Data context

- What is the value of a given variable in this context?

```
(let ((x 3)                          { }
      (y 4))
  (+ (let ((x (+ y 5)))              {x=3, y=4}
       (* x y))                      {x=9, y=4}
     x))
```

The concept of environment sets the data context of program execution.

Introduction: Control Context

Control context

- Where do the program execute in the next?

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

```
  (fact 3)                          [ ]
= (* 3 (fact 2))                    (*3 [ ])
= (* 3 (* 2 (fact 1)))              (*3 (* 2 [ ]))
= (* 3 (* 2 (* 1 (fact 0))))        (*3 (*2 (* 1 [ ])))
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
```

$= (* \ 3 \ 2)$
$= 6$

Introduction: Control Context (Cont.)

Control context

- Where do the program execute in the next?

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

```
  (fact-iter 3)                    [ ]
= (fact-iter-acc 3 1)              [ ]
= (fact-iter-acc 2 3)              [ ]
= (fact-iter-acc 1 6)              [ ]
= (fact-iter-acc 0 6)              [ ]
= 6
```

Introduction: Control Context (Cont.)

In the two examples of control contexts, *it is evaluation of operands, not the calling of procedures, that makes the control context grow.*

Expressions with a hole (e.g., [ ] , (*3 [ ]), and (*3 (*2 [ ]))) are called *continuation*, which captures control context.

Continuation is also called as the rest of the computation.

Introduction: Control Context (Cont.)

Continuation as an abstraction of the control context

Concepts	Program Execution Context	
Environment	Data Context	in Ch.3
Continuation	Control Context	in this chapter

In this chapter, we will learn how to track and manipulate control contexts.

Our central tool will be the data type of continuations.

5.1 A Continuation-Passing Interpreters

In our new interpreter, the major procedures such as value-of will take a third parameter, the *continuation*,

- which is intended to be an abstraction of the control context where each expression is evaluated.

Beginning with an interpreter of LETREC (Fig. 5.1), the main goal is to rewrite it so that no calls to value-of builds control context.

The continuation of an expression represents a procedure that takes the result of the expression and completes the (rest of) computation.

Two interfaces for continuations (Cont)

- apply-cont : Cont $\times$ ExpVal $\rightarrow$ FinalAnswer (i.e., ExpVal)
- end-cont $\in$ Cont : (apply-cont end-cont val) $=$ val

5.1 A Continuation-Passing Interpreters (Cont.)

value-of/k : Exp $\times$ Env $\times$ Cont $\rightarrow$ FinalAnswer

- (lambda (exp env cont) ...)

The main goal is to rewrite the LETREC interpreter (Fig. 5.1) with value-of/k so that no call to value-or builds control context.

For zero? expression:

```
(define value-of
 (lambda (exp env)
  (cases expression exp
   (zero?-exp (exp1)
     (let ((num1 (expval-> (value-of exp1 env))))
       (if (zero? num1) (bool-val #t) (bool-val #f)))
          ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

For zero? expression (cont.):

```
(define value-of/k
 (lambda (exp env cont)
  (cases expression exp
   (zero?-exp (exp1)
     (value-of/k exp1 env
        (zero1-cont cont))
              ... )))

(apply-cont (zero1-cont cont) val)
= (apply-cont cont
     (bool-val
       (zero? (expval->num val))))
```

5.1 A Continuation-Passing Interpreters (Cont.)

Const, variable, and procedure are expressions that are already values. No further evaluation is needed to get a result value from each.

For const, variable, and procedure:

```
(define value-of
 (lambda (exp env)
  (cases expression exp
    (const-exp (num) (num-val num))
    (var-exp (var) (apply-env env var))
    (proc-exp (var body)
       (proc-val (procedure var body env)))
    ... )))
```

The expression value gets returned to the control context by applying cont to itself.

5.1 A Continuation-Passing Interpreters (Cont.)

For const, variable, and procedure:

```
(define value-of/k
 (lambda (exp env cont)
  (cases expression exp
    (const-exp (num)
      (apply-cont cont (num-val num)))
    (var-exp (var)
      (apply-cont cont (apply-env env var)))
    (proc-exp (var body)
       (apply-cont cont (proc-val
         (procedure var body env))) )
    ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

Control context will grow when the let bound expression is evaluated.

- This is where a continuation is needed to be introduced.

For let expression:

```
(define value-of
 (lambda (exp env)
  (cases expression exp
   (let-exp (var exp1 body)
     (let ((val1 (value-of exp1 env)))
       (value-of body (extend-env var val1 env))))
        ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

let-exp-cont is introduced.

- it takes a value for the let variable, and
- it continues to evaluate the body of the let expression

For let expression (cont.):

```
(define value-of/k
 (lambda (exp env cont)
  (cases expression exp
   (let-exp (var exp1 body)
     (value-of/k exp1 env
      (let-exp-cont var body env cont))
         ... )))

(apply-cont (let-exp-cont var body env cont) val)
= (value-of/k body (extend-env var val env) cont)
```

5.1 A Continuation-Passing Interpreters (Cont.)

Question. Do we need to introduce a continuation for letrec expression as well?

For letrec expression:

```
(define value-of
 (lambda (exp env)
  (cases expression exp
   (letrec-exp (p-name b-var p-body letrec-body)
     (value-of letrec-body
       (extend-env-rec p-name b-var p-body env)))
          ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

value-of for letrec expression is in tail position where control context does not grow.

For letrec expression:

```
(define value-of/k
 (lambda (exp env cont)
  (cases expression exp
   (letrec-exp (p-name b-var p-body letrec-body)
     (value-of/k letrec-body
       (extend-env-rec p-name b-var p-body env)) cont)
         ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

For diff-exp and call-exp, more than one continuation is needed to be introduced.

Question. Explain the reason.

For difference expression:

```
(define value-of
 (lambda (exp env)
  (cases expression exp
   (diff-exp (exp1 exp2)
     (value-of letrec-body
       (let ((num1 (expval->num (value-of exp1 env)))
             (num2 (expval->num (value-of exp2 env))))
         (num-val (- num1 num2)))))
       ... )))
```

5.1 A Continuation-Passing Interpreters (Cont.)

For difference expression:

```
(define value-of/k
 (lambda (exp env cont)
  (cases expression exp
   (diff-exp (exp1 exp2)
     (value-of/k exp1 env
        (diff1-cont exp2 env cont)))  ... )))

(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env (diff2-cont val1 cont))
(apply-cont (diff2-cont val1 cont) val2)
= (num-val (- num1 num2))
```

Question. Explain the relationship among the three continuations above: cont, (diff1-cont exp2 env cont), and (diff2-cont val1 cont).

5.1 A Continuation-Passing Interpreters (Cont.)

Trace examples using value-of/k in Page 149, 150, and152.

with an example, -(-(44, 11) , 3)

For procedure call expression, explain what continuations are needed to introduced.

- (call-exp (rator rand))

5.2 A Trampolined Interpreters

In most procedural languages, writing the interpreter is difficult because the implementation adds to the control context (the stack) on every procedure call instead of growing control context only when necessary.

Question. Recall that when it is necessary to grow control context.

Question. Read the textbook to understand two reasons that most procedural languages are designed to add to the control context on every procedure call.

A solution to resolve the difficulty in writing interpreters

- Trampolining

5.2 A Trampolined Interpreters (Cont.)

Trampolining : To avoid having an unbounded chain of procedure calls, we break the chain by having one of the procedures in the interpreter actually return a zero-argument procedure.

value-of/k is redesigned to return either an ExpVal or a zero-argument procedure

- Bounce ::= ExpVal + () → Bounce

```
(define trampoline
 (lambda (bounce)
  (if (expval? bounce)
   bounce
   (trampoline (bounce)))))
```

5.2 A Trampolined Interpreters (Cont.)

The *contracts* of all the procedures in the trampolined interpreter are reviewed to be satisfied.

- In value-of-program :
  (trampoline (value-of/k exp (init-env) (end-cont)))
- value-of/k :
  Exp $\times$ Env $\times$ Cont $\rightarrow$ FinalAnswer
  changes to Exp $\times$ Env $\times$ Cont $\rightarrow$ Bounce
- apply-cont :
  Cont $\times$ ExpVal $\rightarrow$ FinalAnswer
  changes to Cont $\times$ ExpVal $\rightarrow$ Bounce
- and so on

5.3 An Imperative Interpreter

Assignment to shared variables can somtimes be used in place of binding. (c.f. Sec. 4.2.3)

For example, we rewrite the following mutually recursive functions even and odd

```
letrec
 even(x) = if zero?(x) then 1 else (odd sub1 (x))
 odd(x) = if zero?(x) then 0 else (even sub1 (x))
in (odd 13)
```

into one with a shared variable x in place of passing it as an argument.

This rewritten program will then be shown to produce the same trace as the original program does.

5.3 An Imperative Interpreter (Cont.)

Version 1:

```
letrec
 even(x) = if zero?(x) then 1 else (odd sub1 (x))
 odd(x) = if zero?(x) then 0 else (even sub1 (x))
in (odd 13)
```

Version 2:

```
let x = 0 in
letrec  even() = if zero?(x) then 1
                 else let d = set x = sub1 (x)
                      in (odd)
        odd() = if zero?(x) then 0
                 else let d = set x = sub1 (x)
                      in (even)
in let d = set x = 13 in (odd)
```

5.3 An Imperative Interpreter (Cont.)

Version 2:

```
let x = 0 in
letrec  even() = if zero?(x) then 1
                 else let d = set x = sub1 (x)
                      in (odd)
        odd() = if zero?(x) then 0
                else let d = set x = sub1 (x)
                     in (even)
in let d = set x = 13 in (odd)
```

Version 3:

```
      x = 13;
      goto odd;
even: if (x=0) then return(1) else { x = x-1; goto odd; }
odd: if (x=0) then return(0) else { x = x-1; goto even; }
```

5.3 An Imperative Interpreter (Cont.)

In version 1, the procedure bodies look for x in the environment whereas version 2 look for it in the store.

Version 1, 2 and 3 all have the same trace of the computation

```
  (odd 13)
= (even 12)
= (odd 11)
...
= (odd 1)
= (even 0)
= 1
```

The 3rd interpretation is the trace of gotos (called a flowchart program), in which we keep track of the location of the program counter and the contents of the register x.

5.3 An Imperative Interpreter (Cont.)

This transformation works only because the original code the calls to even and odd do not grow any control context; they are tail calls.

A 0-argument tail call is the same as a jump!

If a group of procedures call each other only by tail calls,
then we can translate the calls to use assignment instead of binding,
and then we can translate such an assignment program into a flowchart program.

5.3 An Imperative Interpreter (Cont.)

A group of procedures

- (value-of/k exp evn cont)
- (apply-cont cont val)
- (apply-procedure/k proc1 val cont)

Five global registers

- exp, env, cont, val, proc1

Then we can systematically go through each of our four procedures (including value-of-program) and perform this transformation.

- Each of the three procedures above will be replaced by a zero-argument one. (value-of/k), (apply-cont), (apply-procedure/k)

Each call to one of these procedures will be replaced by code:

- It shares the value of each actual parameter in the corresponding register and then invokes the new zero-argument proceudre.

5.3 An Imperative Interpreter (Cont.)

```
(define value-of/k (lambda (exp env cont)
  (cases expression exp
    (const-exp (num)
      (apply-cont cont (num-val num))) ... )))
```

$\Rightarrow$

```
(define value-of/k (lambda ()
    (cases expression exp
      (const-exp (num)
        (set! cont cont)
        (set! val (num-val num))
        (apply-cont)) ... )))
```

The result of this translation is shown in Figures 5.11- 5.14.

5.3 An Imperative Interpreter (Cont.)

Registerization: a process to translate a continuation-passing interpreter into an imperative language that supports gotos.

5.4 Exceptions

So far we have used continuations only to manage the ordinary flow of control in our languages.

But continuations allow us to alter the control context as well.

Exception handling will be added to our defined language.

Expression :: = try Expression catch (Identifier) Expression

try-exp (exp1 var handler-exp)

Expression ::= raise Expression

raise-exp (exp)

5.4 Exceptions (Cont.)

Examples using exceptions

```
let list-index =
  proc (str)
    letrec inner (lst)
      = if null? (lst)
        then raise (``List Index Failed'')
        else if string-equal? (car (lst), str)
               then 0
               else -((inner cdr (lst)), -1)
```

5.4 Exceptions (Cont.)

Examples using exceptions (cont.)

```
let find-member-number =
    proc (member-name)
      ...
        try ((list-index member-name) member-list)
        catch (exn)
            raise (``Can't Find Member Number'')
```

5.4 Exceptions (Cont.)

Implementing this exception handling mechanism using the continuation-passing interpreter is straightforward.

1) When the body of the try expression returns normally:

```
(try-exp (exp1 var handler-exp)
   (value-of/k exp1 env
        (try-cont var handler-exp env cont)))
```

```
(apply-cont (try-cont var handler-exp env cont) val)
= (apply-cont cont val)
```

5.4 Exceptions (Cont.)

2) When the body of the try expression returns an exception (val):

```
(value-of-k (lambda (exp env cont)
    ...
    (raise-exp (exp1)
               (value-of/k exp1 env
                 (raise1-cont cont))) ... ))
...
(apply-cont (raise1-cont cont) val)
= (apply-handler val cont)
```

We attempt to find the closest exception handler and applies it.

5.4 Exceptions (Cont.)

2) When the body of the try expression returns an exception (val): (Cont.)

```
(apply-cont (raise1-cont cont) val)
= (apply-handler val cont)
```

We attempt to find the closest exception handler and applies it.

```
(define apply-handler (lambda (val cont)
  (cases continuation cont
    (try-cont (var handler-exp saved-env saved-cont)
      (value-of/k handler-exp
        (extend-env var val saved-env) saved-cont))
```

```
(end-cont () report-uncaught-exception))
(diff1-cont (exp2 saved-env saved-cont)
  (apply-handler val saved-cont))
... ))
```

5.4 Exceptions (Cont.)

Example:

```
let index = proc (n)
  letrec inner (lst) = if null? (lst)
    then raise 99 else
     if zero? (- (car (lst), n)) then 0 else
      -((inner cdr (lst)), -1)
  in proc (lst)
     try (inner lst) catch (x) -1
in ((index 5) list (2, 3))
```

5.5 Threads

Threads : multiple computations running in the same address space.

Goal: To modify our interpreter to simulate the execution of multi-threaded programs communicating through a single shared memory.

The entire computation in our system consists of a pool of threads.
- Each thread is running, runnable, or blocked.
- The runnable threads will be kept on a ready queue.
- Exactly on thread is running
- Threads that are not ready to be run are blocked

Threads are scheduled for execution by a scheduler with the ready queue and a timer
- A thread completes a certain number of steps, and then
- it is interrupted and put back on the ready queue, and
- a new thread is selected from the ready queue to run

5.5 Threads (Cont.)

The three main topics on threads

- Creation of a new thread (spawn)
- No busy-wait (mutex, wait, signal)
- No interferences by synchronization (mutex, wait, signal)

5.5 Threads (Cont.)

A new language THREADS extending IMPLICIT-REFS

- new threads are created by a construct called *spawn*

1) Two non-cooperating therads

```
letrec
 noisy (l) = if null?(l) then 0 else
     begin print (car (l)); (noisy cdr (l)) end
in begin
 spawn (proc (d) (noisy [1,2,3,4,5]));
 spawn (proc (d) (noisy [6,7,8,9,10]));
 print (100);
 33
end
```

5.5 Threads (Cont.)

2) Producer and consumer, linked by a buffer

```
let buffer = 0 in
let producer = proc (n)
 letrec
  wait (k) = if zero?(k) then set buffer = n
             else begin print (-(k,-200)); (wait -(k,1)) end
 in (wait 5)
let consumer = proc (d)
 letrec
  busywait (k) = if zero?(buffer) then
     begin print (-(k,-100)); (busywait -(k,-1)) end
   else buffer
 in (busywait 0)
in begin spawn (proc (d) (producer 44)); print (300);
     (consumer 96) end
```

5.5 Threads (Cont.)

Implementation of an interpreter using a scheduler
- cf. a continuation-passing interpreter for IMPLICIT-REFS

The scheduler keeps a state of four values:
- the ready-queue, the-final-answer,
- the-max-time-slice, the-time-remaining

The scheduler provides six interfaces:
- initialize-scheduler!, place-on-ready-queue!, run-next-thread,
- time-expired?, decrement-timer!, set-final-answer!

Q. Explain where each of the six interfaces is used.

A thread is represented by a Scheme procedure of no arguments that returns an expressed value:
- Thread $= () \rightarrow$ ExpVal

5.5 Threads (Cont.)

A spawn expression evaluates its argument in a continuation which, when executed, places a new thread on the ready queue and continues.

```
(value-of/k exp env cont
 (case expression exp
  ...
  (spawn-exp (exp1)
   (value-of/k exp env (spawn-cont cont)) ... )))
```

The definition of spawn-cont is in the next slide.

5.5 Threads (Cont.)

The new thread is placed on the ready queue and we continue our computation (saved-cont).

```
(define apply-cont
 (lambda (cont val)
  ...
  (case continuation cont
  ...
   (spawn-cont (saved-cont)
    (let ((proc1 (expval-proc val)))
     (place-on-ready-queue!
      (lambda ()
       (apply-procedure/k proc1
        (num-val 28)
```

```
    (end-subthread-cont))))
  (apply-cont saved-cont (num-val 73)))) ... )))
```

5.5 Threads (Cont.)

Two new ending continuations:

- the main thread runs with a continuation that records the value of the main thread as the final answer.
- When the subthread finishes, we do not report its value.

```
(end-main-thread-cont ()
 (set-final-answer! val)
 (run-next-thread))

 (end-subthread-cont ()
  (run-next-thread))
```

5.5 Threads (Cont.)

The entire system starts with value-of-program which first initializes the scheduler and then runs an expression with end-main-thread-cont.

```
(define value-of-program
 (lambda (timeslice pgm)
  (initialize-store!)
  (initialize-scheduler! timesclie)
  (cases program pgm
   (a-program (exp1)
    (value-of/k
     exp1
     (init-env)
     (end-main-thread-cont))))))
```

5.5 Threads (Cont.)

apply-cont is modified to decrement the timer each time it is called.
If the timer has expired, the then current computation is suspended.

```
(define apply-cont
 (lambda (cont val)
  (if (time-expreid?)

   (begin
    (place-on-ready-queue!
     (lambda () (apply-cont cont val)))
    (run-next-thread))

   (begin
    (decrement-timer!)
```

```
(case continuation cont
  ...    )))))
```

5.5 Threads: Synchronization

Shared variables are an unreliable method of communication because serveral threads may try to write to the same variable.

```
let x = 0
in let incr_x = proc (id)
                  proc (dummy)
                   set x = -(x, 1)
in begin
  spawn ((incr_x 100));
  spawn ((incr_x 200));
  spawn ((incr_x 300))
end
```

Q. Interferences among threads

5.5 Threads: Synchronization (Cont.)

A synchronization facility to ensure that

- no race condition: no interferences occur and
- no busy-wait as in the consumer example

Instead of busy-wait, the program should be able to put itself to sleep and be awakened when the procedure has inserted a value in the shared buffer.

5.5 Threads: Synchronization (Cont.)

Mutex (short for mutual exclusion) or binary semaphore

A mutex is either open or closed. It contains a queue of threads that are waiting for the mutex to become open.

Three operations on mutex

- mutex: to create an initial open mutex
- wait: to try to access to a mutex given as its argument
- signal: to release a mutex given as its argument

It is guaranteed that only one thread can execute between a successive pair of calls to wait and signal.

- critical region

5.5 Threads: Synchronization (Cont.)

A safe counter using a mutex

```
let x = 0
in let mut = mutex ()
in let incr_x = proc (id)
                  proc (dummy)
                   begin
                    wait(mut);
                    set x = -(x, 1)
                    signal (mut)
                   end
in begin
  spawn ((incr_x 100));
  spawn ((incr_x 200));
  spawn ((incr_x 300))
end
```

5.5 Threads: Synchronization (Cont.)

A mutex is modeled as two references: one to its state (either open or closed)

```
(define-datatype mutex mutex?
  (a-mutex
    (ref-to-closed? reference?)
    (ref-to-wait-queue reference?)))
```

- ref-to-closed? : #t or #f

- ref-to-wait-queue : a list of procedures representing threads

Mutexs are made one of expressed values.

The three new operations on mutex

- mutex, wait, and signal over the two references (See P.187)

5.5 Threads: Synchronization (Cont.)

In value-of/k, three new cases for mutex-exp, wait-exp, and signal-exp:

```
(value-of/k exp env cont
 (case expression exp
   ...
   (mutex-exp ()
    (apply-cont cont (mutex-val (new-mutex))))
   (wait-exp (exp1)
    (value-of/k exp1 env (wait-cont cont)))
   (signal-exp (exp1)
    (value-of/k exp1 env (signal-cont cont)))
   ... ))
```

The definitions of wait-cont and signal-cont are in the next slides.

5.5 Threads: Synchronization (Cont.)

```
(wait-exp (exp1)
 (value-of/k exp1 env (wait-cont cont)))

(define apply-cont
 (lambda (cont val)
  ...
   (case continuation cont

    (wait-cont (saved-cont)
     (wait-for-mutex
      (expval-mutex val)
      (lambda () (apply-cont saved-cont (num-val 52)))))
    ... )))
```

See the def. of wait-for-mutex in Figure 5.22.

5.5 Threads: Synchronization (Cont.)

```
(signal-exp (exp1)
 (value-of/k exp1 env (signal-cont cont)))

(define apply-cont
 (lambda (cont val)
  ...
   (case continuation cont

    (signal-cont (saved-cont)
     (singla-mutex
      (expval-mutex val)
      (lambda () (apply-cont saved-cont (num-val 53)))))
   ... )))
```

See the def. of signal-mutex in Figure 5.22.

5.5 Threads: Synchronization (Cont.)

mutex

- It takes no arguments and creates an initially open mutex.

wait

- If the mutex is closed, the current thread gets into the mutex's wait queue, and is suspended. ($blocked$)
- If the mutex is open, it becomes closed and the current thread continues to run.

5.5 Threads: Synchronization (Cont.)

signal

- If the mutex is closed and no threads wait on the queue, then the mutex becomes open and the current thread proceeds.
- If the mutex is closed and some threads are on the queue, then one of them is put on the scheduler's ready queue, and the mutex remains closed.
- The thread that executed the signal continues to compute