Konzepte von Programmiersprachen

Chapter 5: Continuation-Passing Interpreters

Stefan Wehr

Universität Freiburg

8. Juni 2009

Motivation

- *Continuations:* abstraction of the notion of control context (*Environments:* abstraction of data contexts)
- Support for control operators:
  - Exceptions
  - Threads
  - Coroutines
  - `call/cc`
- Introducing continuations is a common transformation in compilers:
  - Makes control-flow explicit
  - Introduces names for intermediate results

What is a *control context*?

Intuition (not a formal definition)

- the “rest” of the program
- “things” that happen after evaluating the current expression

Outline

- Rewrite interpreter for the LETREC language in continuation-passing style (CPS): **make control context explicit by passing around a continuation**
- Use a trick that allows our CPS interpreter to be written in languages without support for tail calls
- Perform a translation of our CPS interpreter to use jumps instead of procedure calls
- Use the CPS interpreter to implement exceptions
- (Use the CPS interpreter to implement threads)

A Continuation-Passing Interpreter

Design principles for the CPS (continuation-passing style) interpreter:

- No call to `value-of` grows the control context of the underlying scheme interpreter
  ⇒ all calls to `value-of` must be tail calls
- Make the control context explicit using continuations

What’s a tail call, anyway?

Definition

- A procedure call is in *tail position* if it is the last operation of the calling procedure.
- A *tail call* is a procedure call in tail position.
- A *tail-recursive* procedure is a procedure whose recursive calls are all tail calls.

Tail calls give raise the *tail call optimization*:

- Do not push the stack frame of the procedure being called on the stack frame of the calling procedure.
- Instead, pop the stack frame of the calling procedure before performing the tail call.
- Tail-recursive procedures under tail call optimization require only constant stack space.

Recursive definition of the factorial function

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

Calculating with `fact`:

```
  (fact 3)
= (* 3 (fact 2))
= (* 3 (* 2 (fact 1)))
= (* 3 (* 2 (* 1 (fact 0))))
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6
```

Tail recursive definition of the factorial function

```
(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)))))
```

Calculating with `fact-iter`:

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

What’s a continuation, anyway?

Definition

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

An interface for continuations

```
FinalAnswer = ExpVal
apply-cont : Cont * ExpVal -> FinalAnswer
```

CPS interpreter: `value-of-program`

```
;; value-of-program : Program -> FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))

;; value-of/k : Exp * Env * Cont -> FinalAnswer
```

Specification of `apply-cont` (partial)

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "End of computation.~%") val))
```

CPS interpreter: `value-of/k` (1/6)

```
;; value-of/k : Exp * Env * Cont -> FinalAnswer
(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))))
      (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))
      ...
```

CPS interpreter: `value-of/k` (2/6)

```
(zero?-exp (exp1)
  (value-of/k exp1 env
    (zero1-cont cont)))
```

Specification of `apply-cont` (continued, still partial)

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

CPS interpreter: `value-of/k` (3/6)

```
(let-exp (var exp1 body)
  (value-of/k exp1 env
    (let-exp-cont var body env cont)))
```

Specification of `apply-cont` (continued, still partial)

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

CPS interpreter: `value-of/k` (4/6)

```
(if-exp (exp1 exp2 exp3)
  (value-of/k exp1 env
    (if-test-cont exp2 exp3 env cont)))
```

Specification of `apply-cont` (continued, still partial)

```
(apply-cont (if-test-cont exp2 exp3 env cont)
            val)
= (if (expval->bool val)
      (value-of/k exp2 env cont)
      (value-of/k exp3 env cont))
```

CPS interpreter: `value-of/k` (5/6)

```
(diff-exp (exp1 exp2)
  (value-of/k exp1 env
    (diff1-cont exp2 env cont)))
```

Specification of `apply-cont` (continued, still partial)

```
(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env (diff2-cont val1 cont))

(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont (num-val (- num1 num2))))
```

CPS interpreter: `value-of/k` (6/6)

```
      (call-exp (rator rand)
        (value-of/k rator env
          (rator-cont rand env cont)))
))) ;; end definition of value-of/k
```

Specification of `apply-cont` (continued)

```
(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env (rand-cont val1 cont))

(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))
```

CPS interpreter: `apply-procedure/k`

```
;; apply-procedure/k : Proc * ExpVal * Cont
;;                   -> FinalAnswer
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```

Examples

1. `(value-of/k`

   `<<-(-(44,11),3)>>`

   ρ_0

   `#(struct:end-cont))`

2. `(value-of/k <<(`*`exp1 exp2`*`)>>` ρ_1 *`cont1`*`)`

⇒ see blackboard

Implementing the specification for continuations

Two options:

- Procedural representation
- Data structure representation

Procedural representation

```
;; Cont = ExpVal -> FinalAnswer
;; end-cont : () -> Cont
(define end-cont
  (lambda ()
    (lambda (val)
      (begin (eopl:printf "End of computation.~%")
             val))))
;; zero1-cont : Cont -> Cont
(define zero1-cont
  (lambda (cont)
    (lambda (val)
      (apply-cont cont
        (bool-val (zero? (expval->num val)))))))
...

;; apply-cont : Cont * ExpVal -> FinalAnswer
(define apply-cont
  (lambda (cont v) (cont v)))
```

Data structure representation

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (cont continuation?))
  ...
)
;; apply-cont : Cont * ExpVal -> FinalAnswer
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf "End of computation.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val (zero? (expval->num val)))))
      ...   )))
```

A trampolined CPS interpreter

Problems when porting the CPS interpreter to a procedural language:

- No first-class functions
  Solution: use data structure representation for continuations
- No support for tail calls
  ⇒ danger of stack overflow because procedure calls only return when the CPS interpreter ends
  Solution: "Trampolined interpreter" (explained next)

Skeleton of our CPS interpreter

```
;; value-of-program : Program -> FinalAnswer
(define value-of-program
  (... (value-of/k ...) ...))
;; value-of/k : Exp * Env * Cont -> FinalAnswer
(define value-of/k
  (... (value-of/k ...)
       (apply-cont ...) ...))
;; apply-cont : Cont * ExpVal -> FinalAnswer
;; (data structure representation)
(define apply-cont
  (... (value-of/k ...)
       (apply-cont ...)
       (apply-procedure/k ...) ...))
;; apply-procedure/k : Proc * ExpVal * Cont
;;                     -> FinalAnswer
(define apply-procedure/k
  (... (value-of/k ...) ...))
```

Example (CPS interpreter, tail calls)

`value-of-program`

Example (CPS interpreter, tail calls)

`value-of/k`

Example (CPS interpreter, tail calls)

`apply-cont`

Example (CPS interpreter, tail calls)

`apply-procedure/k`

Example (CPS interpreter, tail calls)

`value-of/k`

Example (CPS interpreter, tail calls)

`apply-cont`

Example (CPS interpreter, tail calls)

`apply-procedure/k`

Example (CPS interpreter, tail calls)

`value-of/k`

Example (CPS interpreter, tail calls)

…

Example (CPS interpreter, no tail calls)

`value-of-program`

Example (CPS interpreter, no tail calls)

```
value-ok/k
value-of-program
```

Example (CPS interpreter, no tail calls)

```
apply-cont
value-ok/k
value-of-program
```

Example (CPS interpreter, no tail calls)

```
apply-procedure/k
   apply-cont
   value-ok/k
value-of-program
```

Example (CPS interpreter, no tail calls)

```
  value-ok/k
apply-procedure/k
  apply-cont
  value-ok/k
value-of-program
```

Example (CPS interpreter, no tail calls)

```
    apply-cont
    value-ok/k
apply-procedure/k
    apply-cont
    value-ok/k
 value-of-program
```

Example (CPS interpreter, no tail calls)

```
apply-procedure/k
    apply-cont
    value-ok/k
apply-procedure/k
    apply-cont
    value-ok/k
 value-of-program
```

Example (CPS interpreter, no tail calls)

```
value-of/k
apply-procedure/k
apply-cont
value-ok/k
apply-procedure/k
apply-cont
value-ok/k
value-of-program
```

Example (CPS interpreter, no tail calls)

```
...
value-of/k
apply-procedure/k
apply-cont
value-ok/k
apply-procedure/k
apply-cont
value-ok/k
value-of-program
```

Trampolined interpreter

- Observation: An unbounded chain of procedure calls always involves calls of `apply-procedure/k`.
- Break the chain inside `apply-procedure/k`:
  - `apply-procedure/k` should not call `value-of/k` directly
  - `apply-procedure/k` returns a zero-argument procedure that calls `value-of/k`
- A *trampoline* procedure then invokes this zero-argument procedure to keep the system going.

Example (Trampolined CPS interpreter, no tail calls)

`value-of-program`

Example (Trampolined CPS interpreter, no tail calls)

```
value-ok/k
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
apply-cont
value-ok/k
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

`apply-procedure/k`

`apply-cont`

`value-ok/k`

`value-of-program`

Example (Trampolined CPS interpreter, no tail calls)

```
trampoline
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
value-of/k
trampoline
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
apply-cont
value-of/k
trampoline
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
apply-procedure/k
    apply-cont
    value-of/k
    trampoline
 value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
trampoline
value-of-program
```

Example (Trampolined CPS interpreter, no tail calls)

```
...
trampoline
value-of-program
```

Skeleton of the trampolined CPS interpreter

```
;; value-of-program : Program -> FinalAnswer
(define value-of-program
  (... (trampoline (value-of/k ...)) ...))
;; value-of/k : Exp ⋆ Env ⋆ Cont -> Bounce
(define value-of/k
  (... (value-of/k ...)
       (apply-cont ...) ...))
;; apply-cont : Cont ⋆ ExpVal -> Bounce
;; (data structure representation)
(define apply-cont
  (... (value-of/k ...)
       (apply-cont ...)
       (apply-procedure/k ...) ...))
;; apply-procedure/k : Proc ⋆ ExpVal ⋆ Cont -> Bounce
(define apply-procedure/k
  (... (lambda () (value-of/k ...)) ...))
```

trampoline

```
;; trampoline : Bounce -> FinalAnswer
(define trampoline
  (lambda (bounce)
    (if (expval? bounce)
        bounce
        (trampoline (bounce)))))
```

trampoline

```
;; trampoline : Bounce -> FinalAnswer
(define trampoline
  (lambda (bounce)
    (if (expval? bounce)
        bounce
        (trampoline (bounce)))))
```

$$\texttt{Bounce} = \texttt{ExpVal} \cup \big(\texttt{()} \rightarrow \texttt{Bounce}\big)$$

An interpreter with jumps

- Goal: Get rid off procedure calls in the interpreter, use jumps instead
- Insight I: Use shared variables instead of procedure parameters
- Insight II: A zero-argument tail call is the same as a jump

Example with procedure parameters

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

Example without procedure parameters

```
let x = 0
in letrec
     even() = if zero?(x) then 1
              else begin
                     set x = -(x,1);
                     (odd)
                   end
    odd()  = if zero?(x) then 0
             else begin
                    set x = -(x,1);
                    (even)
                  end
in begin
     set x = 13;
     (odd)
   end
```

Example with jumps

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

Skeleton of our CPS interpreter

```
;; value-of-program : Program -> FinalAnswer
(define value-of-program
  (... (value-of/k ...) ...))
;; value-of/k : Exp * Env * Cont -> FinalAnswer
(define value-of/k
  (... (value-of/k ...)
       (apply-cont ...) ...))
;; apply-cont : Cont * ExpVal -> FinalAnswer
;; (data structure representation)
(define apply-cont
  (... (value-of/k ...)
       (apply-cont ...)
       (apply-procedure/k ...) ...))
;; apply-procedure/k : Proc * ExpVal * Cont
;;                    -> FinalAnswer
(define apply-procedure/k
  (... (value-of/k ...) ...))
```

Introducing “registers”

Relevant procedures of the CPS interpreter:

- `(value-of/k exp env cont)`
- `(apply-cont cont val)`
- `(apply-procedure/k proc val cont)`

⇒ We need five global registers:

- `reg_exp`
- `reg_env`
- `reg_cont`
- `reg_val`
- `reg_proc`

Rewriting the interpreter

Systematically replace fragments such as

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

by

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

Things to watch out for

- Register stay often unchanged from one procedure call to another
  ⇒ omit redundant assignments such as
  `(set! reg_cont reg_cont)`
- Local bindings must not shadow global registers
  ⇒ the prefix `reg_` for register names takes care of that
- Attention is needed if the same registers is used more than once in a procedure call
  ⇒ does not occur in our example

Skeleton of the interpreter with jumps (1/2)

```
;; reg_exp : Exp
(define reg_exp 'uninitialized)

;; reg_env : Env
(define reg_env 'uninitialized)

;; reg_cont : Cont
(define reg_cont 'uninitialized)

;; reg_val : ExpVal
(define reg_val 'uninitialized)

;; reg_proc : Proc
(define reg_proc 'uninitialized)
```

Skeleton of the interpreter with jumps (2/2)

```
;; value-of-program : Program -> FinalAnswer
(define value-of-program
  (... (value-of/k) ...))

;; value-of/k : () -> FinalAnswer
(define value-of/k
  (... (value-of/k)
       (apply-cont) ...))

;; apply-cont : () -> FinalAnswer
(define apply-cont
  (... (value-of/k)
       (apply-cont)
       (apply-procedure/k) ...))

;; apply-procedure/k : () -> FinalAnswer
(define apply-procedure/k
  (... (value-of/k) ...))
```

Exceptions

Two new productions:

Expression ::= `try` *Expression* `catch` `(`*Identifier*`)` *Expression*

`try-exp (exp1 var handler-exp)`

Expression ::= `raise` *Expression*

`raise-exp (exp)`

Semantics:

- `raise` e evaluates e and then raises an exception with that value.
- `try` e_1 `catch` `(`x`)` e_2 first evaluates e_1.
  - If no exception occurs while evaluating e_1, then the result of e_1 is the result of the whole expression.
  - If an exception occurs while evaluating e_1, the exception value is bound to x and the handler e_2 is evaluated.

Implementing exceptions with continuations

```
;; value-of/k : Exp * Env * Cont -> FinalAnswer
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      ...
      (try-exp (exp1 var handler-exp)
        (value-of/k exp1 env
          (try-cont var handler-exp env cont)))
      (raise-exp (exp1)
        (value-of/k exp1 env
          (raise1-cont cont))))))
```

Specification of `apply-cont` (extended)

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

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

`apply-handler`

`(apply-handler val cont)` searches for the closest exception handler in `cont` and applies it to `val`

```
;; apply-handler : ExpVal * Cont -> FinalAnswer
(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))
      ...)))
```

Note: The implementation relies on the data structure representation of continuations. It is also possible to implement `apply-handler` with a procedural representation of continuations (see EOPL, exercise 5.41).