Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

The MIT Press. (c) 2008. Copying Prohibited.

Chapter 6: Continuation-Passing Style

In chapter 5, we took an interpreter and rewrote it so that all of the major procedure calls were *tail calls*. By doing so, we guaranteed that the interpreter uses at most a bounded amount of control context at any one time, no matter how large or complex a program it is called upon to interpret. This property is called *iterative control behavior*.

We achieved this goal by passing an extra parameter, the *continuation*, to each procedure. This style of programming is called *continuation-passing style* or *CPS*, and it is not restricted to interpreters.

In this chapter we develop a systematic method for transforming any procedure into an equivalent procedure whose control behavior is iterative. This is accomplished by converting it into continuation-passing style.

6.1 Writing Programs in Continuation-Passing Style

We can use CPS for other things besides interpreters. Let's consider an old favorite, the factorial program:

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

A continuation-passing version of factorial would look something like

```
(define fact
  (lambda (n)
    (fact/k n (end-cont))))

(define fact/k
  (lambda (n cont)
    (if (zero? n)
      (apply-cont cont 1)
      (fact/k (- n 1) (fact1-cont n cont)))))
```

where

```
(apply-cont (end-cont) val) = val

(apply-cont (fact1-cont n cont) val)
= (apply-cont cont (* n val))
```

In this version, all the calls to fact/k and apply-cont are in tail position and therefore build up no control context.

We can implement these continuations as data structures by writing

```
(define-datatype continuation continuation?
  (end-cont)
  (fact1-cont
    (n integer?)
    (cont continuation?)))

(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont () val)
      (fact1-cont (saved-n saved-cont)
        (apply-cont saved-cont (* saved-n val))))))
```

We can transform this program in many ways. We could, for example, registerize it, as shown in figure 6.1.

```
(define n 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)

(define fact
  (lambda (arg-n)
    (set! cont (end-cont))
    (set! n arg-n)
    (fact/k)))

(define fact/k
  (lambda ()
    (if (zero? n)
      (begin
        (set! val 1)
        (apply-cont))
      (begin
        (set! cont (fact1-cont n cont))
        (set! n (- n 1))
        (fact/k)))))

(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont () val)
      (fact1-cont (saved-n saved-cont)
        (set! cont saved-cont)
        (set! n saved-n)
        (apply-cont)))))
```

Figure 6.1: `fact/k` registerized

We could even trampoline this version, as shown in figure 6.2. If we did this in an ordinary imperative language, we would of course replace the trampoline by a proper loop.

```
(define n 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)
(define pc 'uninitialized)

(define fact
  (lambda (arg-n)
    (set! cont (end-cont))
    (set! n arg-n)
    (set! pc fact/k)
    (trampoline!)
    val))

(define trampoline!
  (lambda ()
    (if pc
      (begin
        (pc)
        (trampoline!)))))

(define fact/k
  (lambda ()
    (if (zero? n)
      (begin
        (set! val 1)
        (set! pc apply-cont))
      (begin
        (set! cont (fact1-cont n cont))
        (set! n (- n 1))
        (set! pc fact/k)))))

(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont ()
        (set! pc #f))
      (fact1-cont (saved-n saved-cont)
        (set! cont saved-cont)
        (set! n saved-n)
        (set! pc apply-cont)))))
```

Figure 6.2: `fact/k` registerized and trampolined

However, our primary concern in this chapter will be what happens when we use a procedural representation, as we did in figure 5.2. Recall that in the procedural representation, a continuation is represented by its action under apply-cont. The procedural representation looks like

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fact1-cont
  (lambda (n saved-cont)
    (lambda (val)
      (apply-cont saved-cont (* n val)))))

(define apply-cont
  (lambda (cont val)
    (cont val)))
```

We can do even better by taking each call to a continuation-builder in the program and replacing it by its definition. This transformation is called *inlining*, because the definitions are expanded in-line. We also inline the calls to apply-cont, so instead of writing (apply-cont cont val), we'll just write (cont val).

If we inline all the uses of continuations in this way, we get

```
(define fact
  (lambda (n)
    (fact/k n (lambda (val) val))))

(define fact/k
  (lambda (n cont)
    (if (zero? n)
      (cont 1)
      (fact/k (- n 1) (lambda (val) (cont (* n val)))))))
```

We can read the definition of fact/k as:

> If n is zero, send 1 to the continuation. Otherwise, evaluate fact of n – 1 in a continuation that calls the result val, and then sends to the continuation the value (* n val).

The procedure fact/k has the property that (fact/k *n g*) = (*g n*!). This is easy to show by induction on *n*. For the base step, when *n* = 0, we calculate

```
(fact/k 0 g) = (g 1) = (g (fact 0))
```

For the induction step, we assume that (fact/k *n g*) = (*g n*!), for some value of *n* and try to show that (fact/k (*n* + 1) *g*) = (*g* (*n* + 1)!). To do this, we calculate:

```
(fact/k n + 1 g)
= (fact/k n (lambda (val) (g (* n + 1 val))))
= ((lambda (val) (g (* n + 1 val)))   (by the induction hypothesis)
   (fact n))
= (g (* n + 1 (fact n)))
= (g (fact n + 1))
```

This completes the induction.

Here the *g* appears as a context argument, as in section 1.3, and the property that (fact/k *n g*) = (*g n*!) serves as the independent specification, following our principle of *No Mysterious Auxiliaries*.

Now let's do the same thing for the Fibonacci sequence fib. We start with

```
(define fib
  (lambda (n)
    (if (< n 2)
      1
      (+
        (fib (- n 1))
        (fib (- n 2))))))
```

Here we have two recursive calls to fib, so we will need an end-cont and two continuation-builders, one for each argument, just as we did for difference expressions in section 5.1.

```
(define fib
  (lambda (n)
    (fib/k n (end-cont))))

(define fib/k
  (lambda (n cont)
    (if (< n 2)
      (apply-cont cont 1)
      (fib/k (- n 1) (fib1-cont n cont)))))

(apply-cont (end-cont) val) = val
```

```
(apply-cont (fib1-cont n cont) val1)
= (fib/k (- n 2) (fib2-cont val1 cont))

(apply-cont (fib2-cont val1 cont) val2)
= (apply-cont cont (+ val1 val2))
```

In the procedural representation we have

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fib1-cont
  (lambda (n cont)
    (lambda (val1)
      (fib/k (- n 2) (fib2-cont val1 cont)))))

(define fib2-cont
  (lambda (val1 cont)
    (lambda (val2)
      (apply-cont cont (+ val1 val2)))))

(define apply-cont
  (lambda (cont val)
    (cont val)))
```

If we inline all the uses of these procedures, we get

```
(define fib
  (lambda (n)
    (fib/k n (lambda (val) val))))

(define fib/k
  (lambda (n cont)
    (if (< n 2)
      (cont 1)
      (fib/k (- n 1)
        (lambda (val1)
          (fib/k (- n 2)
            (lambda (val2)
              (cont (+ val1 val2)))))))))
```

As we did for factorial, we can read this definition as

> If n < 2, send 1 to the continuation. Otherwise, work on n – 1 in a continuation that calls the result val1 and then works on n – 2 in a continuation that calls the result val2 and then sends (+ val1 val2) to the continuation.

It is easy to see, by the same reasoning we used for fact, that for any *g*, (fib/k *n g*) = (*g* (fib *n*)). Here is an artificial example that extends these ideas.

```
(lambda (x)
  (cond
    ((zero? x) 17)
    ((= x 1) (f x))
    ((= x 2) (+ 22 (f x)))
    ((= x 3) (g 22 (f x)))
    ((= x 4) (+ (f x) 33 (g y)))
    (else (h (f x) (- 44 y) (g y)))))
```

becomes

```
(lambda (x cont)
  (cond
    ((zero? x) (cont 17))
    ((= x 1) (f x cont))
    ((= x 2) (f x (lambda (v1) (cont (+ 22 v1)))))
    ((= x 3) (f x (lambda (v1) (g 22 v1 cont))))
    ((= x 4) (f x (lambda (v1)
                    (g y (lambda (v2)
                           (cont (+ v1 33 v2))))))
    (else (f x (lambda (v1)
                 (g y (lambda (v2)
                        (h v1 (- 44 y) v2 cont)))))))
```

where the procedures f, g, h, j, and p have been similarly transformed.

- In the (zero? x) line, we return 17 to the continuation.
- In the (= x 1) line, we call f tail-recursively.
- In the (= x 2) line, we call f in an operand position of an addition.
- In the (= x 3) line, we call f in an operand position of a procedure call.
- In the (= x 4) line, we have two procedure calls in operand positions in an addition.
- In the else line, we have two procedure calls in operand position inside another procedure call.

From these examples, we can see a pattern emerging.

The CPS Recipe

To convert a program to continuation-passing style

1. Pass each procedure an extra parameter (typically cont or k).
2. Whenever the procedure returns a constant or variable, return that value to the continuation instead, as we did with (cont 7) above.
3. Whenever a procedure call occurs in a tail position, call the procedure with the same continuation cont.
4. Whenever a procedure call occurs in an operand position, evaluate the procedure call in a new continuation that gives a name to the result and continues with the computation.

These rules are informal, but they illustrate the patterns.

Exercise 6.1 [*] Consider figure 6.2 without (set! pc fact/k) in the definition of fact/k and without (set! pc apply-cont) in the definition of apply-cont. Why does the program still work?

Exercise 6.2 [*] Prove by induction on *n* that for any *g*, (fib/k *n g*) = (*g* (fib *n*)).

Exercise 6.3 [*] Rewrite each of the following Scheme expressions in continuation-passing style. Assume that any unknown functions have also been rewritten in CPS.

1. (lambda (x y) (p (+ 8 x) (q y)))
2. (lambda (x y u v) (+ 1 (f (g x y) (+ u v))))
3. (+ 1 (f (g x y) (+ u (h v))))
4. (zero? (if a (p x) (p y)))
5. (zero? (if (f a) (p x) (p y)))
6. (let ((x (let ((y 8)) (p y)))) x)
7. (let ((x (if a (p x) (p y)))) x)

Exercise 6.4 [**] Rewrite each of the following procedures in continuation-passing style. For each procedure, do this first using a data-structure representation of continuations, then with a procedural representation, and then with the inlined procedural representation. Last, write the registerized version. For each of these four versions, test to see that your implementation is tail-recursive by defining end-cont by

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "End of computation.~%")
    (eopl:printf "This sentence should appear only once.~%")
    val)
```

as we did in chapter 5.

1. remove-first (section 1.2.3).
2. list-sum (section 1.3).

3. occurs-free? (section 1.2.4).
4. subst (section 1.2.5).

Exercise 6.5 [*] When we rewrite an expression in CPS, we choose an evaluation order for the procedure calls in the expression. Rewrite each of the preceding examples in CPS so that all the procedure calls are evaluated from right to left.

Exercise 6.6 [*] How many different evaluation orders are possible for the procedure calls in (lambda (x y) (+ (f (g x)) (h (j y))))? For each evaluation order, write a CPS expression that calls the procedures in that order.

Exercise 6.7 [**] Write out the procedural and the inlined representations for the interpreter in figures 5.4, 5.5, and 5.6.

Exercise 6.8 [***] Rewrite the interpreter of section 5.4 using a procedural and inlined representation. This is challenging because we effectively have two observers, apply-cont and apply-handler. As a hint, consider modifying the recipe on page 6.1 so that we add to each procedure two extra arguments, one representing the behavior of the continuation under apply-cont and one representing its behavior under apply-handler.

Sometimes we can find clever representations of continuations. Let's reconsider the version of fact with the procedural representation of continuations. There we had two continuation builders, which we wrote as

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fact1-cont
  (lambda (n cont)
    (lambda (val) (cont (* n val)))))

(define apply-cont
  (lambda (cont val)
    (cont val)))
```

In this system, all a continuation does is multiply its argument by some number. (end-cont) multiplies its argument by 1, and if *cont* multiplies its value by *k*, then (fact1 *n cont*) multiplies its value by *k* * *n*.

So every continuation is of the form (lambda (val) (* *k* val)). This means we could represent such a continuation simply by its lone free variable, the number *k*. In this representation we would have

```
(define end-cont
  (lambda ()
    1))

(define fact1-cont
  (lambda (n cont)
    (* cont n)))

(define apply-cont
  (lambda (cont val)
    (* cont val)))
```

If we inline these definitions into our original definition of fact/k, and use the property that (* *cont* 1) = *cont*, we get

```
(define fact
  (lambda (n)
    (fact/k n 1)))

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

But this is just the same as fact-iter (page 139)! So we see that an accumulator is often just a representation of a continuation. This is impressive. Quite a few classic program optimizations turn out to be instances of this idea.

Exercise 6.9 [*] What property of multiplication makes this program optimization possible?

Exercise 6.10 [*] For list-sum, formulate a succinct representation of the continuations, like the one for fact/k above.

6.2 Tail Form

In order to write down a program for converting to continuation-passing style, we need to identify the input and output languages. For our input language, we choose the language LETREC, augmented by having multiargument procedures and multideclaration letrec expressions. Its grammar is shown in figure 6.3. We call this language CPS-IN. To distinguish the expressions of this language from those of our output language, we call these *input expressions*.

$$\mathit{Program} ::= \mathit{InpExp}$$
$$\boxed{\texttt{a-program (exp1)}}$$

$$\mathit{InpExp} ::= \mathit{Number}$$
$$\boxed{\texttt{const-exp (num)}}$$

$$\mathit{InpExp} ::= \texttt{-}(\mathit{InpExp}\ \texttt{,}\ \mathit{InpExp})$$
$$\boxed{\texttt{diff-exp (exp1 exp2)}}$$

$$\mathit{InpExp} ::= \texttt{zero?}(\mathit{InpExp})$$
$$\boxed{\texttt{zero?-exp (exp1)}}$$

$$\mathit{InpExp} ::= \texttt{if}\ \mathit{InpExp}\ \texttt{then}\ \mathit{InpExp}\ \texttt{else}\ \mathit{InpExp}$$
$$\boxed{\texttt{if-exp (exp1 exp2 exp3)}}$$

$$\mathit{InpExp} ::= \mathit{Identifier}$$
$$\boxed{\texttt{var-exp (var)}}$$

$$\mathit{InpExp} ::= \texttt{let}\ \mathit{Identifier}\ \texttt{=}\ \mathit{InpExp}\ \texttt{in}\ \mathit{InpExp}$$
$$\boxed{\texttt{let-exp (var exp1 body)}}$$

$$\mathit{InpExp} ::= \texttt{letrec}\ \{\mathit{Identifier}\ (\{\mathit{Identifier}\}^{*(,)})\ \texttt{=}\ \mathit{InpExp}\}^{*}\ \texttt{in}\ \mathit{InpExp}$$
$$\boxed{\texttt{letrec-exp (p-names b-varss p-bodies letrec-body)}}$$

$$\mathit{InpExp} ::= \texttt{proc}\ (\{\mathit{Identifier}\}^{*(,)})\ \mathit{InpExp}$$
$$\boxed{\texttt{proc-exp (vars body)}}$$

$$\mathit{InpExp} ::= (\mathit{InpExp}\ \{\mathit{InpExp}\}^{*})$$
$$\boxed{\texttt{call-exp (rator rands)}}$$

Figure 6.3: Grammar for CPS-IN

To define the class of possible outputs from our CPS conversion algorithm, we need to identify a subset of CPS-IN in which procedure calls never build any control context.

Recall our principle from chapter 5:

> It is evaluation of operands, not the calling of procedures, that makes the control context grow.

Thus in

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

it is the position of the call to fact *as an operand* that requires the creation of a control context. By contrast, in

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

none of the procedure calls is in operand position. We say these calls are in *tail position* because their value is the result of the whole call. We refer to them as *tail calls*.

We can also recall the Tail Calls Don’t Grow Control Context principle:

> **Tail Calls Don’t Grow Control Context**
>
> If the value of exp_1 is returned as the value of exp_2, then exp_1 and exp_2 should run in the same continuation.

We say that an expression is in *tail form* if every procedure call, and every expression containing a procedure call, is in tail position. This condition implies that no procedure call builds control context.

Hence in Scheme

```
(if (zero? x) (f y) (g z))
```

is in tail form, as is

```
(if b
  (if (zero? x) (f y) (g z))
  (h u))
```

but

```
(+
  (if (zero? x) (f y) (g z))
  37)
```

is not in tail form, since the if expression, which contains a procedure call, is not in tail position.

In general, we must understand the meaning of a language in order to determine its tail positions. A subexpression in tail position has the property that if it is evaluated, its value immediately becomes the value of the entire expression. An expression may have more than one tail position. For example, an if expression may choose either the true or the false branch. For a subexpression in tail position, no information need be saved, and therefore no control context need be built.

The tail positions for CPS-IN are shown in figure 6.4. The value of each subexpression in tail position could become the value of the entire expression. In the continuation-passing interpreter, the subexpressions in operand positions are the ones that require building new continuations. The subexpressions in tail position are evaluated in the same continuation as the original expression, as illustrated on page 152.

```
zero?(O)
-(O, O)
if O then T else T
let Var = O in T
letrec {Var ({Var}*(,)) = T}* in T
proc ({Var}*(,)) T
(OO ... O )
```

Figure 6.4: Tail and operand positions in CPS-IN. Tail positions are marked with T. Operand positions are marked with O.

We use this distinction to design a target language CPS-OUT for our CPS conversion algorithm. The grammar for this language is shown in figure 6.5. This grammar defines a subset of CPS-IN, but with a different grammar. Its production names always begin with cps-, so they will not be confused with the production names in CPS-IN.

Program ::= *TfExp*

`a-program (exp1)`

SimpleExp ::= *Number*

`const-exp (num)`

SimpleExp ::= *Identifier*

`var-exp (var)`

SimpleExp ::= `-(`*SimpleExp* `,` *SimpleExp*`)`

`cps-diff-exp (simple1 simple2)`

SimpleExp ::= `zero?(`*SimpleExp*`)`

`cps-zero?-exp (simple1)`

SimpleExp ::= `proc` `(`$\{\mathit{Identifier}\}^{*}$`)` *TfExp*

`cps-proc-exp (vars body)`

TfExp ::= *SimpleExp*

`simple-exp->exp (simple-exp1)`

TfExp ::= `let` *Identifier* `=` *SimpleExp* `in` *TfExp*

`cps-let-exp (var simple1 body)`

TfExp ::= `letrec` $\{\mathit{Identifier}$ `(`$\{\mathit{Identifier}\}^{*(,)}$`)` `=` $\mathit{TfExp}\}^{*}$ `in` *TfExp*

`cps-letrec-exp (p-names b-varss p-bodies body)`

TfExp ::= `if` *SimpleExp* `then` *TfExp* `else` *TfExp*

`cps-if-exp (simple1 body1 body2)`

TfExp ::= `(`*SimpleExp* $\{\mathit{SimpleExp}\}^{*}$`)`

`cps-call-exp (rator rands)`

Figure 6.5: Grammar for CPS-OUT

The new grammar has two nonterminals, *SimpleExp* and *TfExp*. It is designed so that expressions in *SimpleExp* are guaranteed never to contain any procedure calls, and so that expressions in *TfExp* are guaranteed to be in tail form.

Expressions in *SimpleExp* are guaranteed to never contain any procedure calls, so they correspond roughly to simple straight-line code, and for our purposes we consider them too simple to require any use of the control stack. Simple expressions include proc expressions, since a proc expression returns immediately with a procedure value, but the body of that procedure must be in tail form.

A continuation-passing interpreter for tail-form expressions is shown in figure 6.6. Since procedures in this language take multiple arguments, we use extend-env* from exercise 2.10 to create multiple bindings, and we similarly extend extend-env-rec to get extend-env-rec*.

```
value-of/k : TfExp × Env × Cont → FinalAnswer
(define value-of/k
  (lambda (exp env cont)
    (cases tfexp exp
      (simple-exp->exp (simple)
        (apply-cont cont
          (value-of-simple-exp simple env)))
      (let-exp (var rhs body)
       (let ((val (value-of-simple-exp rhs env)))
          (value-of/k body
            (extend-env (list var) (list val) env)
            cont)))
      (letrec-exp (p-names b-varss p-bodies letrec-body)
        (value-of/k letrec-body
          (extend-env-rec** p-names b-varss p-bodies env)
          cont))
      (if-exp (simple1 body1 body2)
        (if (expval->bool (value-of-simple-exp simple1 env))
          (value-of/k body1 env cont)
          (value-of/k body2 env cont)))
      (call-exp (rator rands)
        (let ((rator-proc
                (expval->proc
                  (value-of-simple-exp rator env)))
              (rand-vals
                (map
                  (lambda (simple)
                    (value-of-simple-exp simple env))
                  rands)))
          (apply-procedure/k rator-proc rand-vals cont))))))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure/k
  (lambda (proc1 args cont)
    (cases proc proc1
      (procedure (vars body saved-env)
        (value-of/k body
          (extend-env* vars args saved-env)
          cont)))))
```

Figure 6.6: Interpreter for tail-form expressions in CPS-OUT.

In this interpreter, all the recursive calls are in tail position (in Scheme), so running the interpreter builds no control context in Scheme. (This isn't quite true: the procedure value-of-simple-exp (exercise 6.11) builds control context in Scheme, but that can be fixed (see exercise 6.18).)

More importantly, the interpreter creates no new continuations. The procedure value-of/k takes one continuation argument and passes it unchanged in every recursive call. So we could easily have removed the continuation argument entirely.

Of course, there is no completely general way of determining whether the control behavior of a procedure is iterative or not. Consider

```
(lambda (n)
  (if (strange-predicate? n)
    (fact n)
    (fact-iter n)))
```

This procedure is iterative only if strange-predicate? returns false for all sufficiently large values of n. But it is not always possible to determine the truth or falsity of this condition, even if it is possible to examine the code of strange-predicate?. Therefore the best we can hope for is to make sure that no procedure call in the program will build up control context, whether or not it is actually executed.

Exercise 6.11 [*] Complete the interpreter of figure 6.6 by writing value-of-simple-exp.

Exercise 6.12 [*] Determine whether each of the following expressions is simple.

1. -((f -(x,1)),1))
2. (f -(-(x,y),1)))
3. if zero?(x) then -(x,y) else -(-(x,y),1))
4. let x = proc (y) (y x) in -(x,3))
5. let f = proc (x) x in (f 3))

Exercise 6.13 [*] Translate each of these expressions in CPS-IN into continuation-passing style using the CPS recipe on page 200 above. Test your transformed programs by running them using the interpreter of figure 6.6. Be sure that the original and transformed versions give the same answer on each input.

1. removeall.

```
letrec
 removeall(n,s) =
  if null?(s)
  then emptylist
  else if number?(car(s))
       then if equal?(n,car(s))
            then (removeall n cdr(s))
            else cons(car(s),
                      (removeall n cdr(s)))
       else cons((removeall n car(s)),
                 (removeall n cdr(s)))
```

2. occurs-in?.

```
letrec
 occurs-in?(n,s) =
  if null?(s)
  then 0
  else if number?(car(s))
       then if equal?(n,car(s))
            then 1
            else (occurs-in? n cdr(s))
       else if (occurs-in? n car(s))
            then 1
            else (occurs-in? n cdr(s))
```

3. remfirst. This uses occurs-in? from the preceding example.

```
letrec
 remfirst(n,s) =
  letrec
   loop(s) =
    if null?(s)
    then emptylist
    else if number?(car(s))
         then if equal?(n,car(s))
              then cdr(s)
              else cons(car(s),(loop cdr(s)))
         else if (occurs-in? n car(s))
              then cons((remfirst n car(s)),
                        cdr(s))
              else cons(car(s),
                        (remfirst n cdr(s)))
in (loop s)
```

4. depth.

```
letrec
 depth(s) =
  if null?(s)
  then 1
  else if number?(car(s))
       then (depth cdr(s))
       else if less?(add1((depth car(s))),
                     (depth cdr(s)))
            then (depth cdr(s))
            else add1((depth car(s)))
```

5. depth-with-let.

```
letrec
 depth(s) =
  if null?(s)
  then 1
  else if number?(car(s))
       then (depth cdr(s))
       else let dfirst = add1((depth car(s)))
                drest = (depth cdr(s))
            in if less?(dfirst,drest)
               then drest
               else dfirst
```

6. map.

```
letrec
 map(f, l) = if null?(l)
             then emptylist
             else cons((f car(l)),
                        (map f cdr(l)))
 square(n) = *(n,n)
in (map square list(1,2,3,4,5))
```

7. fnlrgtn. This procedure takes an n-list, like an s-list (page 9), but with numbers instead of symbols, and a number n and returns the first number in the list (in left-to-right order) that is greater than n. Once the result is found, no further elements in the list are examined. For example,

```
(fnlrgtn list(1,list(3,list(2),7,list(9)))
 6)
```

   finds 7.

8. every. This procedure takes a predicate and a list and returns a true value if and only if the predicate holds for each list element.

```
letrec
 every(pred, l) =
  if null?(l)
  then 1
  else if (pred car(l))
       then (every pred cdr(l))
       else 0
 in (every proc(n)greater?(n,5) list(6,7,8,9))
```

Exercise 6.14 [*] Complete the interpreter of figure 6.6 by supplying definitions for value-of-program and apply-cont.

Exercise 6.15 [*] Observe that in the interpreter of the preceding exercise, there is only one possible value for cont. Use this observation to remove the cont argument entirely.

Exercise 6.16 [*] Registerize the interpreter of figure 6.6.

Exercise 6.17 [*] Trampoline the interpreter of figure 6.6.

Exercise 6.18 [**] Modify the grammar of CPS-OUT so that a simple diff-exp or zero?-exp can have only a constant or variable as an argument. Thus in the resulting language value-of-simple-exp can be made nonrecursive.

Exercise 6.19 [**] Write a Scheme procedure tail-form? that takes the syntax tree of a program in CPS-IN, expressed in the grammar of figure 6.3, and determines whether the same string would be in tail form according to the grammar of figure 6.5.

6.3 Converting to Continuation-Passing Style

In this section we develop an algorithm for transforming any program in CPS-IN to CPS-OUT.

Like the continuation-passing interpreter, our translator will *Follow the Grammar*. Also like the continuation-passing interpreter, our translator will take an additional argument that represents a continuation. This additional argument will be a simple expression that represents the continuation.

As we have done in the past, we will proceed from examples to a specification, and from a specification to a program. Figure 6.7 shows a somewhat more detailed version of the motivating examples, written in Scheme so that they will be similar to those of the preceding section.

```
(lambda (x)
  (cond
    ((zero? x) 17)
    ((= x 1) (f (- x 13) 7))
    ((= x 2) (+ 22 (- x 3) x))
    ((= x 3) (+ 22 (f x) 37))
    ((= x 4) (g 22 (f x)))
    ((= x 5) (+ 22 (f x) 33 (g y)))
    (else (h (f x) (- 44 y) (g y))))))

becomes
```

```
(lambda (x k)
  (cond
    ((zero? x) (k 17))
    ((= x 1) (f (- x 13) 7 k))
    ((= x 2) (k (+ 22 (- x 3) x)))
    ((= x 3) (f x (lambda (v1) (k (+ 22 v1 37)))))
    ((= x 4) (f x (lambda (v1) (g 22 v1 k))))
    ((= x 5) (f x (lambda (v1)
                    (g y (lambda (v2)
                           (k (+ 22 v1 33 v2))))))
    (else (f x (lambda (v1)
                 (g y (lambda (v2)
                        (h v1 (- 44 y) v2 k))))))))
```

Figure 6.7: Motivating examples for CPS conversion (in Scheme)

The first case is that of a constant. Constants are just sent to the continuation, as in the (zero? x) line above.

```
(cps-of-exp n K) = (K n)
```

Here *K* is some simple-exp that denotes a continuation.

Similarly, variables are just sent to the continuation.

```
(cps-of-exp var K) = (K var)
```

Of course, the input and output of our algorithm will be abstract syntax trees, so we should have written the builders for the abstract syntax instead of the concrete syntax, like

```
(cps-of-exp (const-exp n) K)
= (make-send-to-cont K (cps-const-exp n))

(cps-of-exp (var-exp var) K)
= (make-send-to-cont K (cps-var-exp var))
```

where

```
make-send-to-cont : SimpleExp × SimpleExp → TfExp
(define make-send-to-cont
  (lambda (k-exp simple-exp)
    (cps-call-exp k-exp (list simple-exp))))
```

We need the list since in CPS-OUT every call expression takes a list of operands.

We will, however, continue to use concrete syntax in our specifications because the concrete syntax is generally easier to read.

What about procedures? We convert a procedure, like the (lambda (x) …) in figure 6.7, by adding an additional parameter k and converting the body to send its value to the continuation k. This is just what we did in figure 6.7. So

proc (var_1, ..., var_n) *exp*

becomes

proc (var_1, ..., var_n, k) (cps-of-exp *exp* k)

as in the figure. However, this doesn't quite finish the job. Our goal was to produce code that would evaluate the proc expression and send the result to the continuation *K*. So the entire specification for a proc expression is

(cps-of-exp <<proc (var_1, ..., var_n) *exp*>> *K*)
= (*K* <<proc (var_1, ..., var_n, k) (cps-of-exp *exp* k)>>)

Here k is a fresh variable, and *K* is an arbitrary simple expression that denotes a continuation.

What about expressions that have operands? Let us add, for the moment, a sum expression to our language, with arbitrarily many operands. To do this, we add to the grammar of CPS-IN the production

$$Expression ::= +\,(\{InpExp\}^{*(,)})$$
$$\boxed{\texttt{sum-exp (exps)}}$$

and to the grammar of CPS-OUT the production

$$SimpleExp ::= +\,(\{SimpleExp\}^{*(,)})$$
$$\boxed{\texttt{cps-sum-exp (simple-exps)}}$$

This new production preserves the property that no procedure call ever appears inside a simple expression.

What are the possibilities for (cps-of-exp «+(exp_1, ..., exp_n)» *K*)? It could be that all of exp_1, …, exp_n are simple, as in the (= x 2) case in figure 6.7. Then the entire sum expression is simple, and we can just pass it to the continuation. We let *simp* denote a simple expression. In this case we can say

```
(cps-of-exp <<+(simp_1, ..., simp_n)>> K)
= (K +(simp_1, ..., simp_n))
```

What if one of the operands is nonsimple? Then we need to evaluate it in a continuation that gives a name to its value and proceeds with the sum, as in the (= x 3) case above. There the second operand is the first nonsimple one. Then our CPS converter should have the property that

```
(cps-of-exp <<+(simp_1, exp_2, simp_3, ..., simp_n)>> K)
= (cps-of-exp exp_2
    <<proc (var_2) (K +(simp_1, var_2, simp_3, ..., simp_n))>>
```

If exp_2 is just a procedure call, then the output will look like the one in the figure. But exp_2 might be more complicated, so we recur, calling cps-of-exp on exp_2 and the larger continuation

```
proc (var_2) (K +(simp_1, var_2, simp_3, ..., simp_n))
```

There might, however, be other nonsimple operands in the sum expression, as there are in the (= x 5) case. So instead of simply using the continuation

```
proc (var_2) (K +(simp_1, var_2, simp_3, ..., simp_n))
```

we need to recur on the later arguments as well. We can summarize this rule as

```
(cps-of-exp <<+(simp_1, exp_2, exp_3, ..., exp_n)>> K)
= (cps-of-exp exp_2
    <<proc (var_2)
       (cps-of-exp <<+(simp_1, var_2, exp_3, ..., exp_n)>> K))
```

Each of the recursive calls to cps-of-exp is guaranteed to terminate. The first call terminates because exp_2 is smaller than the original expression. The second call terminates because its argument is also smaller than the original: var_2 is always smaller than exp_2.

For example, looking at the (= x 5) line and using the syntax of CPS-IN, we have

```
(cps-of-exp <<+((f x), 33, (g y))>> K)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp +(v1, 33, (g y)) K)>>)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(g y)>>
         <<proc (v2)
            (cps-of-exp <<+(v1, 33, v2)>> K)))
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(g y)>>
         <<proc (v2)
            (K <<+(v1, 33, v2)>>)))
= (f x
  proc (v1)
   (g y
    proc (v2)
     (K +(v1, 33, v2))))
```

Procedure calls work the same way. If both the operator and all the operands are simple, then we just call the procedure with a continuation argument, as in the (= x 2) line.

```
(cps-of-exp <<(simp_0 simp_1 ... simp_n)>> K)
= (simp_0 simp_1 ... simp_n K)
```

If, on the other hand, one of the operands is nonsimple, then we must cause it to be evaluated first, as in the (= x 4) line.

```
(cps-of-exp <<(simp_0 simp_1 exp_2 exp_3 ... exp_n)>> K)
= (cps-of-exp exp_2
    <<proc (var_2)
       (cps-of-exp <<(simp_0 simp_1 var_2 exp_3 ... exp_n)>> K)>>)
```

And, as before, the second call to cps-of-exp will recur down the procedure call, calling cps-of-exp for each of the nonsimple arguments, until there are only simple arguments left.

Here is how these rules handle the (= x 5) example, written in CPS-IN.

```
(cps-of-exp <<(h (f x) -(44,y) (g y))>> K)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(h v1 -(44,y) (g y))>> K)>>)
= (f x
   proc (v1)
    (cps-of-exp <<(h v1 -(44,y) (g y))>> K)>>)
= (f  x
   proc (v1)
    (cps-of-exp <<(g y)>>
     <<proc (v2)
        (cps-of-exp <<(h v1 -(44,y) v2)>> K)))
= (f x
   proc (v1)
    (g y
     proc (v2)
      (cps-of-exp <<(h v1 -(44,y) v2)>> K)))
= (f x
   proc (v1)
    (g y
     proc (v2)
      (h v1 -(44,y) v2 K)))
```

The specifications for sum expressions and procedure calls follow a similar pattern: they find the first nonsimple operand and recur on that operand and on the modified list of operands. This works for any expression that evaluates its operands. If complex-exp is some CPS-IN expression that evaluates its operands, then we should have

```
(cps-of-exp (complex-exp simp_0 simp_1 exp_2 exp_3 ... exp_n) K)
= (cps-of-exp exp_2
    <<proc (var_2)
       (cps-of-exp
         (complex-exp simp_0 simp_1 var_2 exp_3 ... exp_n)
        K)>>)
```

where *var_2* is a fresh variable.

The only time that the treatment of sum expressions and procedure calls differs is when the arguments are all simple. In that case, we need to convert each of the arguments to a CPS-OUT simple-exp and produce a tail-form expression with the results.

We can encapsulate this behavior into the procedure cps-of-exps, shown in figure 6.8. Its arguments are a list of input expressions and a procedure builder. It finds the position of the first nonsimple expression in the list, using the procedure list-index from exercise 1.23. If there is such a nonsimple expression, then it is converted in a continuation that gives the value a name (the identifier bound to var) and recurs down the modified list of expressions.

```
cps-of-exps : Listof(InpExp) × (Listof(InpExp) → TfExp) → TfExp
(define cps-of-exps
  (lambda (exps builder)
    (let cps-of-rest ((exps exps))
      cps-of-rest : Listof(InpExp) → TfExp
      (let ((pos (list-index
                   (lambda (exp)
                     (not (inp-exp-simple? exp)))
                   exps)))
        (if (not pos)
          (builder (map cps-of-simple-exp exps))
          (let ((var (fresh-identifier 'var)))
            (cps-of-exp
              (list-ref exps pos)
              (cps-proc-exp (list var)
                (cps-of-rest
                  (list-set exps pos (var-exp var))))))))))

inp-exp-simple? : InpExp → Bool
(define inp-exp-simple?
  (lambda (exp)
    (cases expression exp
      (const-exp (num) #t)
      (var-exp (var) #t)
      (diff-exp (exp1 exp2)
        (and (inp-exp-simple? exp1) (inp-exp-simple? exp2)))
      (zero?-exp (exp1) (inp-exp-simple? exp1))
      (proc-exp (ids exp) #t)
      (sum-exp (exps) (every? inp-exp-simple? exps))
      (else #f))))
```

Figure 6.8: `cps-of-exps`

If there are no nonsimple expressions, then we would like to apply builder to the list of expressions. However, although these expressions are simple, they are still in the grammar of CPS-IN. Therefore we convert each expression to the grammar of CPS-OUT using the procedure cps-of-simple-exp. We then send the list of *SimpleExp*s to builder. (list-set is described in exercise 1.19.)

The procedure inp-exp-simple? takes an expression in CPS-IN and determines whether its string would be parseable as a *SimpleExp*. It uses the procedure every? from exercise 1.24. The expression (every? *pred lst*) returns #t if every element of *lst* satisfies *pred*, and returns #f otherwise.

The code for cps-of-simple-exp is straightforward and is shown in figure 6.9. It also translates the body of a proc-exp into CPS, which is necessary for the output to be a *SimpleExp*.

```
cps-of-simple-exp : InpExp → SimpleExp
usage:   assumes (inp-exp-simple? exp).
(define cps-of-simple-exp
  (lambda (exp)
    (cases expression exp
      (const-exp (num) (cps-const-exp num))
      (var-exp (var) (cps-var-exp var))
      (diff-exp (exp1 exp2)
        (cps-diff-exp
          (cps-of-simple-exp exp1)
          (cps-of-simple-exp exp2)))
      (zero?-exp (exp1)
        (cps-zero?-exp (cps-of-simple-exp exp1)))
      (proc-exp (ids exp)
        (cps-proc-exp (append ids (list 'k%00))
          (cps-of-exp exp (cps-var-exp 'k%00))))
      (sum-exp (exps)
        (cps-sum-exp (map cps-of-simple-exp exps)))
      (else
        (report-invalid-exp-to-cps-of-simple-exp exp)))))
```

Figure 6.9: `cps-of-simple-exp`

We can generate tail-form expressions for sum expressions and procedure calls using cps-of-exps.

```
cps-of-diff-exp : Listof(InpExp) × SimpleExp → TfExp
(define cps-of-sum-exp
  (lambda (exps k-exp)
    (cps-of-exps exps
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-sum-exp simples))))))

cps-of-call-exp : InpExp × Listof(InpExp) × SimpleExp → TfExp
(define cps-of-call-exp
  (lambda (rator rands k-exp)
    (cps-of-exps (cons rator rands)
      (lambda (simples)
        (cps-call-exp
          (car simples)
          (append (cdr simples) (list k-exp)))))))
```

We can now write the rest of our CPS translator (figures 6.10–6.12). It *Follows the Grammar*. When the expression is always simple, as for constants, variables, and procedures, we generate the code immediately using make-send-to-cont. Otherwise, we call an auxiliary procedure. Each auxiliary procedure calls cps-of-exps to evaluate its subexpressions, supplying an appropriate builder to construct the innermost portion of the CPS output. The one exception is cps-of-letrec-exp, which has no immediate subexpressions, so it generates the CPS output directly. Finally, we translate a program by calling cps-of-exps on the whole program, with a builder that just returns the value of the simple.

```
cps-of-exp : InpExp × SimpleExp → TfExp
(define cps-of-exp
  (lambda (exp k-exp)
    (cases expression exp
      (const-exp (num)
        (make-send-to-cont k-exp (cps-const-exp num)))
      (var-exp (var)
        (make-send-to-cont k-exp (cps-var-exp var)))
      (proc-exp (vars body)
        (make-send-to-cont k-exp
          (cps-proc-exp (append vars (list 'k%00))
            (cps-of-exp body (cps-var-exp 'k%00)))))
      (zero?-exp (exp1)
        (cps-of-zero?-exp exp1 k-exp))
      (diff-exp (exp1 exp2)
        (cps-of-diff-exp exp1 exp2 k-exp))
      (sum-exp (exps)
        (cps-of-sum-exp exps k-exp))
      (if-exp (exp1 exp2 exp3)
        (cps-of-if-exp exp1 exp2 exp3 k-exp))
      (let-exp (var exp1 body)
        (cps-of-let-exp var exp1 body k-exp))
      (letrec-exp (p-names b-varss p-bodies letrec-body)
        (cps-of-letrec-exp
          p-names b-varss p-bodies letrec-body k-exp))
      (call-exp (rator rands)
        (cps-of-call-exp rator rands k-exp)))))

cps-of-zero?-exp : InpExp × SimpleExp → TfExp
(define cps-of-zero?-exp
  (lambda (exp1 k-exp)
    (cps-of-exps (list exp1)
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-zero?-exp
            (car simples)))))))
```

Figure 6.10: `cps-of-exp`, part1

```
cps-of-diff-exp : InpExp × InpExp × SimpleExp → TfExp
(define cps-of-diff-exp
  (lambda (exp1 exp2 k-exp)
    (cps-of-exps
      (list exp1 exp2)
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-diff-exp
            (car simples)
            (cadr simples)))))))

cps-of-if-exp : InpExp × InpExp × InpExp × SimpleExp → TfExp
(define cps-of-if-exp
  (lambda (exp1 exp2 exp3 k-exp)
    (cps-of-exps (list exp1)
      (lambda (simples)
        (cps-if-exp (car simples)
          (cps-of-exp exp2 k-exp)
          (cps-of-exp exp3 k-exp))))))

cps-of-let-exp : Var × InpExp × InpExp × SimpleExp → TfExp
(define cps-of-let-exp
  (lambda (id rhs body k-exp)
    (cps-of-exps (list rhs)
      (lambda (simples)
        (cps-let-exp id
          (car simples)
          (cps-of-exp body k-exp))))))

cps-of-letrec-exp :
 Listof(Var) × Listof(Listof(Var)) × Listof(InpExp) × SimpleExp → TfExp
(define cps-of-letrec-exp
  (lambda (p-names b-varss p-bodies letrec-body k-exp)
    (cps-letrec-exp
      p-names
      (map
        (lambda (b-vars) (append b-vars (list 'k%00)))
        b-varss)
      (map
        (lambda (p-body)
          (cps-of-exp p-body (cps-var-exp 'k%00)))
        p-bodies)
      (cps-of-exp letrec-body k-exp))))
```

Figure 6.11: `cps-of-exp`, part 2

```
cps-of-program : InpExp → TfExp
(define cps-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cps-a-program
          (cps-of-exps (list exp1)
            (lambda (new-args)
              (simple-exp->exp (car new-args)))))))))
```

Figure 6.12: cps-of-program

For the following exercises, make sure that your output expressions are in tail form by running them using the grammar and interpreter for CPS-OUT.

Exercise 6.20 [*] Our procedure cps-of-exps causes subexpressions to be evaluated from left to right. Modify cps-of-exps so that subexpressions are evaluated from right to left.

Exercise 6.21 [*] Modify cps-of-call-exp so that the operands are evaluated from left to right, followed by the operator.

Exercise 6.22 [*] Sometimes, when we generate (*K simp*), *K* is already a proc-exp. So instead of generating

```
(proc (var₁) ... simp)
```

we could generate

```
let var₁ = simp
in ...
```

This leads to CPS code with the property that it never contains a subexpression of the form

```
(proc (var) exp₁
 simp)
```

unless that subexpression was in the original expression.

Modify make-send-to-cont to generate this better code. When does the new rule apply?

Exercise 6.23 [**] Observe that our rule for if makes two copies of the continuation *K*, so in a nested if the size of the transformed program can grow exponentially. Run an example to confirm this observation. Then show how this may be avoided by changing the transformation to bind a fresh variable to *K*.

Exercise 6.24 [**] Add lists to the language (exercise 3.10). Remember that the arguments to a list are not in tail position.

Exercise 6.25 [**] Extend CPS-IN so that a let expression can declare an arbitrary number of variables (exercise 3.16).

Exercise 6.26 [**] A continuation variable introduced by cps-of-exps will only occur once in the continuation. Modify make-send-to-cont so that instead of generating

```
let var1 = simp1
in T
```

as in exercise 6.22, it generates $T[simp_1/var_1]$, where the notation $E_1[E_2/var]$ means expression E_1 with every free occurrence of the variable *var* replaced by E_2.

Exercise 6.27 [**] As it stands, cps-of-let-exp will generate a useless let expression. (Why?) Modify this procedure so that the continuation variable is the same as the let variable. Then if exp_1 is nonsimple,

```
(cps-of-exp <<let var1 = exp1 in exp2>> K)
= (cps-of-exp exp1 <<proc (var1) (cps-of-exp exp2 K)>>
```

Exercise 6.28 [*] Food for thought: imagine a CPS transformer for Scheme programs, and imagine that you apply it to the first interpreter from chapter 3. What would the result look like?

Exercise 6.29 [**] Consider this variant of cps-of-exps.

```
(define cps-of-exps
  (lambda (exps builder)
    (let cps-of-rest ((exps exps) (acc '()))
      cps-of-rest : Listof(InpExp) × Listof(SimpleExp) → TfExp
      (cond
        ((null? exps) (builder (reverse acc)))
        ((inp-exp-simple? (car exps))
         (cps-of-rest (cdr exps)
           (cons
             (cps-of-simple-exp (car exps))
             acc)))
        (else
         (let ((var (fresh-identifier 'var)))
           (cps-of-exp (car exps)
             (cps-proc-exp (list var)
               (cps-of-rest (cdr exps)
                 (cons
                   (cps-of-simple-exp (var-exp var))
                   acc)))))))))
```

Why is this variant of cps-of-exp more efficient than the one in figure 6.8?

Exercise 6.30 [**] A call to cps-of-exps with a list of expressions of length one can be simplified as follows:

```
(cps-of-exps (list exp) builder)
= (cps-of-exp/ctx exp (lambda (simp) (builder (list simp))))
```

where

```
cps-of-exp/ctx : InpExp × (SimpleExp → TfExp) → TfExp
(define cps-of-exp/ctx
  (lambda (exp context)
    (if (inp-exp-simple? exp)
      (context (cps-of-simple-exp exp))
      (let ((var (fresh-identifier 'var)))
        (cps-of-exp exp
          (cps-proc-exp (list var)
            (context (cps-var-exp var))))))))
```

Thus, we can simplify occurrences of (cps-of-exps (list ...)), since the number of arguments to list is known. Therefore the definition of, for example, cps-of-diff-exp could be defined with cps-of-exp/ctx instead of with cps-of-exps.

```
(define cps-of-diff-exp
  (lambda (exp1 exp2 k-exp)
    (cps-of-exp/ctx exp1
      (lambda (simp1)
        (cps-of-exp/ctx exp2
          (lambda (simp2)
            (make-send-to-cont k-exp
              (cps-diff-exp simp1 simp2))))))))
```

For the use of cps-of-exps in cps-of-call-exp, we can use cps-of-exp/ctx on the rator, but we still need cps-of-exps for the rands. Remove all other occurrences of cps-of-exps from the translator.

Exercise 6.31 [***] Write a translator that takes the output of cps-of-program and produces an equivalent program in which all the continuations are represented by data structures, as in chapter 5. Represent data structures like those constructed using define-datatype as lists. Since our language does not have symbols, you can use an integer tag in the car position to distinguish the variants of a data type.

Exercise 6.32 [***] Write a translator like the one in exercise 6.31, except that it represents all procedures by data structures.

Exercise 6.33 [***] Write a translator that takes the output from exercise 6.32 and converts it to a register program like the one in figure 6.1.

Exercise 6.34 [**] When we convert a program to CPS, we do more than produce a program in which the control contexts become explicit. We also choose the exact order in which the operations are done, and choose names for each intermediate result. The latter is called *sequentialization*. If we don't care about obtaining iterative behavior, we can sequentialize a program by converting it to *A-normal form* or *ANF*. Here's an example of a program in ANF.

```
(define fib/anf
  (lambda (n)
    (if (< n 2)
      1
      (let ((val1 (fib/anf (- n 1))))
        (let ((val2 (fib/anf (- n 2))))
          (+ val1 val2))))))
```

Whereas a program in CPS sequentializes computation by passing continuations that name intermediate results, a program in ANF sequentializes computation by using let expressions that name all of the intermediate results.

Retarget cps-of-exp so that it generates programs in ANF instead of CPS. (For conditional expressions occurring in nontail position, use the ideas in exercise 6.23.) Then, show that applying the revised cps-of-exp to, e.g., the definition of fib yields the definition of fib/anf. Finally, show that given an input program which is already in ANF, your translator produces the same program except for the names of bound variables.

Exercise 6.35 [*] Verify on a few examples that if the optimization of exercise 6.27 is installed, CPS-transforming the output of your ANF transformer (exercise 6.34) on a program yields the same result as CPS-transforming the program.

6.4 Modeling Computational Effects

Another important use of CPS is to provide a model in which computational effects can be made explicit. A computational effect is an effect like printing or assigning to a variable, which is difficult to model using equational reasoning of the sort used in chapter 3. By transforming to CPS, we can make these effects explicit, just as we did with nonlocal control flow in chapter 5.

In using CPS to model effects, our basic principle is that a simple expression should have no effects. This principle underlies our rule that a simple expression should have no procedure calls, since a procedure call could fail to terminate (which is certainly an effect!).

In this section we study three effects: printing, a store (using the explicit-reference model), and nonstandard control flow.

Let us first consider printing. Printing certainly has an effect:

```
(f print(3) print(4))
```

and

```
(f 1 1)
```

have different effects, even though they return the same answer. The effect also depends on the order of evaluation of arguments; up to now our languages have always evaluated their arguments from left to right, but other languages might not do so.

We can model these considerations by modifying our CPS transformation in the following ways:

- We add to CPS-IN a print expression

InpExp ::= `print (`*InpExp*`)`

$\boxed{\texttt{print-exp (exp1)}}$

We have not written an interpreter for CPS-IN, but the interpreter would have to be extended so that a print-exp prints the value of its argument and returns some value (which we arbitrarily choose to be 38).

- We add to CPS-OUT a printk expression

TfExp ::= `printk (`*SimpleExp*`) ;` *TfExp*

$\boxed{\texttt{cps-printk-exp (simple-exp1 body)}}$

The expression printk(*simp*); *exp* has an effect: it prints. Therefore it must be a *TfExp*, not a *SimpleExp*, and can appear only in tail position. The value of *exp* becomes the value of the entire printk expression, so *exp* is itself in tail position and can be a tfexp. Thus we might write bits of code like

```
proc (v1)
 printk(-(v1,1));
 (f v1 K)
```

To implement this, we add to the interpreter for CPS-OUT the line

```
(printk-exp (simple body)
  (begin
    (eopl:printf "~s~%"
      (value-of-simple-exp simple env))
    (value-of/k body env cont)))
```

- We add to cps-of-exp a line that translates from a print expression to a printk expression. We have arbitrarily decided to have print expression return the value 38. So our translation should be

```
(cps-of-exp <<print(simp1)>> K) = printk(simp1); (K 38))
```

and we use cps-of-exps to take care of the possibility that the argument to print is nonsimple. This gets us to a new line in cps-of-exp that says:

```
(print-exp (rator)
  (cps-of-exps (list rator)
    (lambda (simples)
      (cps-printk-exp
        (car simples)
        (make-send-to-cont k-exp
          (cps-const-exp 38))))))
```

Let us watch this work on a larger example.

```
(cps-of-exp <<(f print((g x)) print(4))>> K)
= (cps-of-exp <<print((g x))>>
    <<proc (v1)
       (cps-of-exp <<(f v1 print(4))>> K)>>)
= (cps-of-exp <<(g x)>>
    <<proc (v2)
       (cps-of-exp <<(print v2)>>
         <<proc (v1)
            (cps-of-exp <<(f v1 print(4))>> K)>>)>>)
= (g x
   proc (v2)
    (cps-of-exp <<(print v2)>>
      <<proc (v1)
         (cps-of-exp <<(f v1 print(4))>> K)))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in (cps-of-exp <<(f v1 print(4)>> K)))

= (g x
   proc (v2)
```

```
    printk(v2);
    let v1 = 38
    in (cps-of-exp <<print(4)>>
         <<proc (v3)
            (cps-of-exp <<(f v1 v3)>> K)>>))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in printk(4);
       let v3 = 38
       in (cps-of-exp <<(f v1 v3)>> K))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in printk(4);
       let v3 = 38
       in (f v1 v3 K))
```

Here, we call g in a continuation that names the result v2. The continuation prints the value of v2 and sends 38 to the next continuation, which binds v1 to its argument 38, prints 4 and then calls the next continuation, which binds v3 to its argument (also 38) and then calls f with v1, v3, and *K*. In this way the sequencing of the different printing actions becomes explicit.

To model explicit references (section 4.2), we go through the same steps: we add new syntax to CPS-IN and CPS-OUT, write new interpreter lines to interpret the new syntax in CPS-OUT, and add new lines to cps-of-exp to translate the new syntax from CPS-IN to CPS-OUT. For explicit references, we will need to add syntax for reference creation, dereference, and assignment.

- We add to CPS-IN the syntax

  InpExp ::= `newref (`*InpExp*`)`
  $\boxed{\texttt{newref-exp (exp1)}}$

  InpExp ::= `deref (`*InpExp*`)`
  $\boxed{\texttt{deref-exp (exp1)}}$

  InpExp ::= `setref (`*InpExp* `,` *InpExp*`)`
  $\boxed{\texttt{setref-exp (exp1 exp2)}}$

- We add to CPS-OUT the syntax

  TfExp ::= `newrefk (`*simple-exp* `,` *simple-exp*`)`
  $\boxed{\texttt{cps-newrefk-exp (simple1 simple2)}}$

  TfExp ::= `derefk (`*simple-exp* `,` *simple-exp*`)`
  $\boxed{\texttt{cps-derefk-exp (simple1 simple2)}}$

  TfExp ::= `setrefk (`*simple-exp* `,` *simple-exp*`) ;` *TfExp*
  $\boxed{\texttt{cps-setrefk-exp (simple1 simple2 body)}}$

  A newrefk expression takes two arguments: the value to be placed in the newly allocated cell, and a continuation to receive a reference to the new location. derefk behaves similarly. Since setref is normally executed for effect only, the design of setrefk follows that of printk. It assigns the value of the second argument to the value of the first argument, which should be a reference, and then executes the third argument tail-recursively.

In this language we would write

```
newrefk(33, proc (loc1)
             newrefk(44, proc (loc2)
                          setrefk(loc1,22);
                          derefk(loc1, proc (val)
                                        -(val,1))))
```

This program allocates a new location containing 33, and binds loc1 to that location. It then allocates a new location containing 44, and binds loc2 to that location. It then sets the contents of location loc1 to 22. Finally, it dereferences loc1, binds the result (which should be 22) to val, and returns the value of -(val,1), yielding 21.

To get this behavior, we add these lines to the interpreter for CPS-OUT.

```
(cps-newrefk-exp (simple1 simple2)
  (let ((val1 (value-of-simple-exp simple1 env))
        (val2 (value-of-simple-exp simple2 env)))
    (let ((newval (ref-val (newref val1))))
      (apply-procedure
        (expval->proc val2)
        (list newval)
        k-exp))))

(cps-derefk-exp (simple1 simple2)
  (apply-procedure
    (expval->proc (value-of-simple-exp simple2 env))
    (list
      (deref
        (expval->ref
          (value-of-simple-exp simple1 env))))
    k-exp))

(cps-setrefk-exp (simple1 simple2 body)
  (let ((ref (expval->ref
               (value-of-simple-exp simple1 env)))
        (val (value-of-simple-exp simple2 env)))
    (begin
      (setref! ref val)
      (value-of/k body env k-exp))))
```

- Finally, we add these lines to cps-of-exp to implement the translation.

```
(newref-exp (exp1)
  (cps-of-exps (list exp1)
    (lambda (simples)
      (cps-newrefk-exp (car simples) k-exp))))

(deref-exp (exp1)
  (cps-of-exps (list exp1)
    (lambda (simples)
      (cps-derefk-exp (car simples) k-exp))))

(setref-exp (exp1 exp2)
  (cps-of-exps (list exp1 exp2)
    (lambda (simples)
      (cps-setrefk-exp
        (car simples)
        (cadr simples)
        (make-send-to-cont k-exp
          (cps-const-exp 23))))))
```

In the last line, we make it appear that a setref returns the value of 23, just like in EXPLICIT-REFS.

Exercise 6.36 [**] Add a begin expression (exercise 4.4) to CPS-IN. You should not need to add anything to CPS-OUT.

Exercise 6.37 [***] Add implicit references (section 4.3) to CPS-IN. Use the same version of CPS-OUT, with explicit references, and make sure your translator inserts allocation and dereference where necessary. As a hint, recall that in the presence of implicit references, a var-exp is no longer simple, since it reads from the store.

Exercise 6.38 [***] If a variable never appears on the left-hand side of a set expression, then it is immutable, and could be treated as simple. Extend your solution to the preceding exercise so that all such variables are treated as simple.

Finally, we come to nonlocal control flow. Let's consider letcc from exercise 5.42. A letcc expression letcc *var* in *body* binds the current continuation to the variable *var*. This binding is in scope in *body*. The only operation on continuations is throw. We use the syntax throw *Expression* to *Expression*, which evaluates the two subexpressions. The second expression should return a continuation, which is applied to the value of the first expression. The current continuation of the throw expression is ignored.

We first analyze these expressions according to the paradigm of this chapter. These expressions are never simple. The body part of a letcc is a tail position, since its value is the value of the entire expression. Since both positions in a throw are evaluated, and neither is the value of the throw (indeed, the throw has no value, since it never returns to its immediate continuation), they are both operand positions.

We can now sketch the rules for converting these two expressions.

```
(cps-of-exp <<letcc var in body>> K)
= let var = K
  in (cps-of-exp body var)

(cps-of-exp <<throw simp1 to simp2>> K)
= (simp2 simp1)
```

We will use cps-of-exps, as usual, to deal with the possibility that the arguments to throw are nonsimple. Here *K* is ignored, as desired.

For this example we do not have to add any syntax to CPS-OUT, since we are just manipulating control structure.

Exercise 6.39 [*] Implement letcc and throw in the CPS translator.

Exercise 6.40 [**] Implement try/catch and throw from section 5.4 by adding them to the CPS translator. You should not need to add anything to CPS-OUT. Instead, modify cps-of-exp to take two continuations: a success continuation and an error continuation.