Continuation-Passing Style

CS 6520, Spring 2002

1 Continuations and Accumulators

In our exploration of machines for ISWIM, we saw how to explicitly track the current continuation for a computation (Chapter 8 of the notes). Roughly, the continuation reflects the control stack in a real machine, including virtual machines such as the JVM. However, accurately reflecting the "memory" use of textual ISWIM requires an implementation different that from that of languages like Java. The ISWIM machine must extend the continuation when evaluating an argument sub-expression, instead of when calling a function. In particular, function calls in tail position require no extension of the continuation. One result is that we can write "loops" using λ and application, instead of a special `for` form.

In considering the implications of tail calls, we saw that some non-loop expressions can be converted to loops. For example, the Scheme program

```
(define (sum n)
  (if (zero? n)
      0
      (+ n (sum (sub1 n)))))
(sum 1000)
```

requires a control stack of depth 1000 to handle the build-up of additions surrounding the recursive call to *sum*, but

```
(define (sum2 n r)
  (if (zero? n)
      r
      (sum2 (sub1 n) (+ r n))))
(sum2 1000 0)
```

computes the same result with a control stack of depth 1, because the result of a recursive call to *sum2* produces the final result for the enclosing call to *sum2*.

Is this magic, or is *sum* somehow special? The *sum* function is slightly special: *sum2* can keep an accumulated total, instead of waiting for all numbers before starting to add them, because addition is commutative.

In contrast, consider a *make-list* function, which takes a positive integer n and creates a list enumerating the integers from n to 1:

```
(define (make-list n)
  (if (zero? n)
      '()
      (cons n (make-list (sub1 n)))))
(make-list 1000)
```

In this case, if we try to accumulate a result as we did for sum,

```
(define (make-list2 n r)
  (if (zero? n)
      r
      (make-list2 (sub1 n) (cons n r))))
(make-list2 1000 '())
```

we get a different result: the list will be in reverse order (i.e., 1 to n). The underlying reason is that cons is not commutative.

We could avoid this problem by creating a function *snoc* to take the place of cons: instead of adding to the front of the list, *snoc* will add to the end:

```
(define (snoc i l)
  (if (null? l)
      (cons i '())
      (cons (car l) (snoc i (cdr l)))))
```

But we haven't achieved anything useful, because *snoc* onto a list of 999 elements requires a control stack of depth 999.

It turns out to be possible to implement *make-list* (or *snoc*) in a purely functional style with a control stack of depth 1. The key is to accumulate something different than a list — something that is not quite the result, but that produces the result when needed.

2 Accumulating a Continuation

The following variation of *make-list* produces the same result as the original, but without using a deep control stack:

```
(define (make-list3 n k)
  (if (zero? n)
      (k '())
      (make-list3 (sub1 n) (lambda (l) (k (cons n l))))))
(make-list3 1000 (lambda (l) l))
```

The accumulator argument k does not accumulate the final result. Instead, it accumulates work to do to produce the final result given an "almost final" result. More precisely, k is a function that takes the list that *make-list3* would otherwise return, and does something to it to get the real final result.

Thus, the base case in *make-list3*, where n is 0, applies k to the empty list instead of returning the result. And the last line of the program, which calls *make-list3* with 1000, provides a k that simply returns the result list.

The key use of k is in the recursive call to *make-list3*. In that case, the recursive call receives a function that consumes the result for $n - 1$, then adds an n to the front. With n added to the front, the result is what *make-list* would have produced, but instead of returning this result directly, the result is provided to k.

In the above program, every call to a non-primitive is in a tail position in the above program. Thus, we need only one control-stack frame to take care of primitive applications in argument positions. Have we changed an $O(n)$ program into an $O(1)$ program? No: although we use $O(1)$ space in the control stack, the k argument will grow on every recursive call:

```
(make-list3 1000 (lambda (l) l))
⇒ (make-list3 999 (lambda (l) ((lambda (l) l) (cons 1000 l))))
⇒ (make-list3 998 (lambda (l) ((lambda (l) ((lambda (l) l) (cons 1000 l))) (cons 999 l))))
⇒ ...
⇒ (make-list3 0 (lambda (l) (... [999 to 2]... (cons 1 l))))
⇒ ((lambda (l) (... [999 to 2]... (cons 1 l))) '())
⇒ (... [999 to 2]... (cons 1 '()))
⇒ ...
```

In fact, k is nothing but the continuation in procedure form! Consequently, it grows on each recursive call in the same way that the control stack used to grow.

3 Continuation-Passing Style

If every program were like *sum2* or *make-list3*, then an evaluator for ISWIM would not need a continuation register at all. By "like *sum2* or *make-list3*", we mean that every function call is in tail position. (Nested primitive operations generate frames in the CEK machine, but the "stack" size for such frames is always limited by the size of the program, and this limit enables an efficient implementation along the lines of the SECD machine.)

The conversion of *make-list* to *make-list3* points to a general conversion strategy for all programs. The result of the conversion is a program in **continuation-passing style** (CPS), which means that the continuation is effectively passed through the program in the form of a procedure argument. More specifically, in a CPS program, each function consumes an extra argument for the continuation, and the function never returns a result directly; instead, it supplies a result to the continuation. Finally, CPS conversion eliminates all nested function calls, so that the continuation argument for a function truly represents the continuation of the function call.

A CPS conversion might be used by a compiler front end to simplify the compiler back end and run-time system. Depending on the application of CPS, it has another advantage: by making the continuation explicit, a program explicitly drop it or use it multiple times. For example, without using an error function, we can escape immediately from a list creation in the innermost part of a CPS recursion:

```
(define (dont-make-list n k)
  (if (zero? n)
      'boom
      (dont-make-list (sub1 n) (lambda (l) (k (cons n l))))))
(dont-make-list 1000 (lambda (l) l))
```

This example demonstrate how we can encode control operations without extending the underlying language.

Informally, to convert a program to CPS:

1. Add an extra argument k to each function.
2. Instead of returning a result in a function, pass the result to k.
3. Lift a nested function call out of its sub-expression by selecting a variable X to replace the function call, wrapping the expression with (**lambda** (X) ...), and providing the resulting (**lambda** X _) as the third argument to the function. For example, convert (add1 (f x)) to (f x (**lambda** (v) (add1 v))). Applications of primitive operations need no lifting.

Applying these rules to *make-list* produces *make-list3*. Here is a more complex source example, the *map* function that takes a function and a list and produces a new list by applying the function to each element of the given list:

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))
```

The CPS conversion proceeds as follows:

- Add a k argument to *map2*:

```
(define (map2 f l k)
  (if (null? l)
      '()
      (cons (f (car l)) (map2 f (cdr l)))))
```

- Pass results to k instead of returning them directly:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (k (cons (f (car l)) (map2 f (cdr l))))))
```

- Lift the (*f* (car *l*)) nested call:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (f (car l) (lambda (v) (k (cons v (map2 f (cdr l))))))))
```

- Lift the (*map2 f* (cdr *l*)) nested call:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (f (car l) (lambda (v) (map2 f (cdr l) (lambda (v2) (k (cons v v2))))))))
```

Note that the kind of function expected by *map2* for *f* is one that accepts a continuation argument in addition to a list element. In general, a CPS conversion changes the meaning of "function" globally so that every function consumes a continuation.

So far, we have been vague about the meaning of "nested". After all, (*f* (car *l*)) is nested in the definition as a whole, but we do not want to lift it out of the definition. Similarly, we did not want to lift (*f* (car *l*)) out of the **if** expression. However, in the following program, we do want to lift (*f* (car *l*)) out of the second **if**:

```
(define (filter f l)
  (if (null? l)
      '()
      (if (f (car l))
          (cons (car l) (filter f (cdr l)))
          (filter f (cdr l)))))
```

In the properly CPS-converted version, we expose the fact that the **if** branch is in the continuation of the (*f* (car *l*)) call:[1]

```
(define (filter2 f l k)
  (if (null? l)
      (k '())
      (f (car l)
         (lambda (v)
           (if v
               (filter2 f (cdr l) (lambda (v2) (cons (car l) v2)))
               (filter2 f (cdr l) (lambda (v2) (k v2)))))))))
```

In general, we lift application expressions out of other applications, out of primitive operations, and out of the test position of **if**. We do not lift past **lambda** (including the implicit one with **define**) or past the branches of an **if** expression. In addition, we perform lifting from left-to-right, which introduces **lambda** that prevent changes to the order of evaluation. For example, lifting (*f* (car *l*)) prevents (*map2 f* (cdr *l*)) from being lifted out into an earlier evaluation position.

[1] To simplify, (**lambda** (*v2*) (*k v2*)) could be replaced with *k*.

3.1 Exercise

Convert the following Scheme program to CPS, where number?, +, cons, car, and cdr are treated as primitive operations.

```
(define (sum-tree stop? t)
  (if (stop? t)
      t
      (+ (sum-tree stop? (car t)) (sum-tree stop? (cdr t)))))
(sum-tree (lambda (x) (number? x)) (cons (cons 1 2) (cons 3 (cons 4 5))))
```

4 Formal CPS Conversion

To formally define a CPS conversion for ISWIM, we first define the set of "simple expressions", S, which do not include function applications. The set of simple expressions also does not include immediate functions (i.e., λ expressions), because they will need to be converted to take a continuation argument.

$$\begin{array}{rcl} S & = & b \\ & | & X \\ & | & (o^n\ S\ \ldots\ S) \end{array}$$

Given the definition of S, we can define a CPS-conversion relation $[\![\_]\!] \bullet [\![\_]\!]$ that maps a source expression and continuation expression to a CPS-converted expression. For example, if the source expression is 4 and the result of the program must is supplied to the continuation k, then

$$[\![4]\!] \bullet [\![k]\!] = (k\ 4)$$

If the expression is $(\lambda y.y)$,

$$[\![(\lambda y.y)]\!] \bullet [\![k]\!] = (k\ (\lambda yk.(k\ y)))$$

In converting a whole program M, we will find $[\![M]\!] \bullet [\![(\lambda x.x)]\!]$. Note that, as a whole program, the converted version of $(\lambda y.y)$ will produce a different kind of function than the original program, because they converted identity function consumes a continuation. However, if we consider the results in terms of $eval_{\mathsf{v}}$, this difference does not matter. Basic constant results are unchanged by the conversion.

The following table defines the $[\![\_]\!] \bullet [\![\_]\!]$ relation:

$$\begin{array}{lcl}
[\![S]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & (M_{\mathsf{k}}\ S) \\
[\![(\lambda X.M)]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & (M_{\mathsf{k}}\ (\lambda X X'.[\![M]\!] \bullet [\![X']\!])) \\
[\![(M_1\ M_2)]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & [\![M_1]\!] \bullet [\![(\lambda X_1.[\![(X_1\ M_2)]\!] \bullet [\![M_{\mathsf{k}}]\!])]\!] \\
\quad \text{if } M_1 \notin S & & \quad \text{where } X_1 \notin \mathcal{FV}(M_2) \cup \mathcal{FV}(M_{\mathsf{k}}) \\
[\![(S_1\ M_2)]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & [\![M_2]\!] \bullet [\![(\lambda X_2.[\![(S_1\ X_2)]\!] \bullet [\![M_{\mathsf{k}}]\!])]\!] \\
\quad \text{if } M_2 \notin S & & \quad \text{where } X_2 \notin \mathcal{FV}(M_{\mathsf{k}}) \\
[\![(S_1\ S_2)]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & (S_1\ S_2\ M_{\mathsf{k}}) \\
[\![(o^n\ S_1 \ldots\ S_{i-1}\ M_i\ M_{i+1} \ldots\ M_n)]\!] \bullet [\![M_{\mathsf{k}}]\!] & = & [\![M_i]\!] \bullet [\![(\lambda X_i.[\![(o^n\ S_1 \ldots\ S_{i-1}\ X_i\ M_{i+1} \ldots\ M_n)]\!] \bullet [\![M_{\mathsf{k}}]\!])]\!] \\
\quad \text{if } M_i \notin S & & \quad \text{where } X_i \notin \mathcal{FV}(M_{i+1}) \cup \ldots \mathcal{FV}(M_n) \cup \mathcal{FV}(M_{\mathsf{k}})
\end{array}$$

By looking at the right-hand sides of the definition, we can easily see that this conversion eliminates all nested function calls. We would also like to know that

$$eval_{\mathsf{v}}(M) \ = \ eval_{\mathsf{v}}([\![M]\!] \bullet [\![(\lambda x.x)]\!])$$

but this fact is difficult to prove.