Buffalo’s Guide to Simple Continuation Passing Style

Preparation –

The goal of CPS is to convert some recursive procedures to non-recursive ones. To start you’ll need:

1. The code of the procedures you intend to change – sometimes it’s just one, other times its one plus some helpers
2. A newly created datatype we will call “continuation”. This datatype will initially have only one flavor – the “all-done” flag which is by tradition called init-k.
3. A newly created procedure we will call (by tradition) apply-k. It takes 2 parameters k – the continuation type, and v – a value. If given init-k, thus function should return v.

Then we will convert the functions one by one. If you can, start with the innermost utility functions so you can test them individually.

For each function…

# Step 1: Places where the result is calculated directly

By directly I mean “without any other CPS function being involved”. Could look really simple like this: #t. Could look a bit more complicated like this: (cons p1 p2).

In this case know what the result of the function should be. But there might be other work that still needs doing stored in the continuation. To do it, call apply-k with the value that would normally be returned e.g.

(cons p1 p2) -> (apply-k k (cons p1 p2))

If you see a place where a result is returned and it involves a non-tail recursive call, you should change it too but realize there is a non-tail call you’ll have to deal with later:

(cons p1 (my-cool-cps-function p2)) -> (apply-k k (cons p1 (my-cool-cps-function p2)))

# Step 2: Places where the result is a CPS procedure call in tail position

In tail position means “I call another CPS procedure and that is the overall result with no additional work required after”. Could look like this: (my-cool-cps-function (cons p1 p2)). But cannot look like this (cons p1 (my-cool-cps-function p2)).

In this case, we can rely on that other CPS function to deal with continuation when the final result is calculated. So we simply pass k along and return that.

(my-cool-cps-function (cons p1 p2)) -> (my-cool-cps-function (cons p1 p2) k)

# Step 3: Places where there is a non-tail CPS procedure call

It could look like this (cons p1 (my-cool-cps-function p2)). It could look like this: (let ((my-val (my-cool-cps-function p2)) some-more-code…). Basically any time you see a call to a cps procedure an it’s not clearly Step 2, it’s this.

If there are several, you can do them in any order. But each requires the following substeps. For my example here, I’ll use this (you can see step 1 has already happened):

(apply-k k (cons p1 (my-cool-cps-function p2)))

3a. Add a new entry in the apply-k case for a newly created created flavor is the current k. Modify the code so that v used as a replacement for the result of the tail call. All other data you need should be added as part of the continuation – and this list will always include the prior continuation k.

(define apply-k

(lambda (k v)

(cases continuation k

...

[stepX (p1 k)

(apply-k k (cons p1 v)))]

3b. Add the new flavor to the continuation type, including the new data you found necessary.

(define-datatype continuation continuation?

...

[stepX (p1 number?) (k continuation?)]

...

3c. Modify the original function code to call the original function in tail position, and pass your new flavor as the k

(my-cool-cps-function p2 (stepX p1 k))

3d. If there was more than one non-tail cps function call, there will be a non-tail call either in the original procedure or in apply-k. Apply 3a-3d again.

An example with multiple non-tail calls would be like this (cons (my-cool-cps-function p1) (my-cool-cps-function p2))

# An Example

# Step 0

We’ll convert slist-find which returns true if a particular symbol is in an slist. I add a k parameter and and continuation type with only init-k, and apply-k that only can handle init-ks.

|  |  |
| --- | --- |
| (define slist-find  (lambda (target slist k)  (cond [(null? slist) #f]  [(symbol? (car slist))  (if (eqv? (car slist) target)  #t  (slist-find target (cdr slist)))]  [else  (if (slist-find target (car slist))  #t  (slist-find target (cdr slist)))]))) | (define-datatype continuation continuation?  [init-k])  (define apply-k  (lambda (k v)  (cases continuation k  [init-k () v]  ))) |

# Step 1

I change the cases where the result is calculated directly. I replace these cases with (apply-k k value)

|  |  |
| --- | --- |
| (define slist-find  (lambda (target slist k)  (cond [(null? slist) (apply-k k #f)]  [(symbol? (car slist))  (if (eqv? (car slist) target)  (apply-k k #t)  (slist-find target (cdr slist)))]  [else  (if (slist-find target (car slist))  (apply-k k #t)  (slist-find target (cdr slist)))]))) | (define-datatype continuation continuation?  [init-k])  (define apply-k  (lambda (k v)  (cases continuation k  [init-k () v]  ))) |

# Step 2

I change the cases where the procedure calls itself or a CPS helper in tail position. In this case you pass along the existing k.

|  |  |
| --- | --- |
| (define slist-find  (lambda (target slist k)  (cond [(null? slist) (apply-k k #f)]  [(symbol? (car slist))  (if (eqv? (car slist) target)  (apply-k k #t)  (slist-find target (cdr slist) k))]  [else  (if (slist-find target (car slist))  (apply-k k #t)  (slist-find target (cdr slist) k))]))) | (define-datatype continuation continuation?  [init-k])  (define apply-k  (lambda (k v)  (cases continuation k  [init-k () v]  ))) |

# Step 3

I change the cases where the procedure calls itself or a CPS helper in non-tail position. For each of these cases, the code after the result is returned must be moved into an apply-k case. Add the data the code needs to run to the datatype. Don’t include the result of the non-tail recursive call – that will be v. In the original code, the non-tail call becomes a tail call, with the new continuation type passed as the final parameter.

|  |  |
| --- | --- |
| (define slist-find  (lambda (target slist k)  (cond [(null? slist) (apply-k k #f)]  [(symbol? (car slist))  (if (eqv? (car slist) target)  (apply-k k #t)  (slist-find target (cdr slist) k))]  [else  (slist-find target  (car slist)  (step1 target  slist  k))]))) | (define-datatype continuation continuation?  [init-k]  [car-find (target symbol?)  (slist list?)  (k continuation?)])  (define apply-k  (lambda (k v)  (cases continuation k  [step1 (target slist k)  (if v  (apply-k k #t)  (slist-find target (cdr slist) k))]  [init-k () v]  ))) |