Recitation 26: Continuations and CPS Conversion

Here are three ways to sum a list of integers: by straightforward recursion, a tail recursive version, and a version using continuations.

```
(* straightforward recursion *)
let rec sum s =
    match s with
        [] -> 0
      | x::xs -> x + sum xs
(* tail recursive *)
let sum s =
  let rec sum' s a =
    match s with
        [] -> a
      | x::xs -> sum' xs (a + x) in
    sum' s 0

(* using continuations *)
let sum s =
  let rec sum' s k =
    match s with
        [] -> k 0
      | x::xs -> sum' xs (fun a -> k (x + a)) in
  sum' s (fun x -> x)
```

In all three cases, we get the same result:

```
- sum [1; 2; 3; 4; 5];
- : int = 1
```

Let's compare these three approaches carefully. The first program, given an input list of `n` integers, will call itself recursively `n` times. On the way down, it does not do any additions, but only evaluates the left argument of the + and remembers this value, and also remembers that an addition must be performed after the sum of the tail has been calculated. This is called a **deferred operation**. It then calls itself recursively on the tail of the input list. The program continues to call itself recursively on successive tails, remembering all the deferred operations and the evaluated left-hand-sides on the runtime stack, until it gets to the empty list, at which point it returns 0. Thereafter, on the way back up out of the recursion, at each level it performs the deferred addition and returns the result to its caller. Thus on input $[x_1,x_2,\ldots,x_n]$, the elements are added in the order

$$(x_1 + (x_2 + (\ldots + (x_{n-1} + (x_n + 0))\ldots))),$$

from right to left.

In the second version, we define an auxiliary function `sum'` that takes an extra argument `a`, an accumulated result. Here the additions are performed on the way *down* in the recursion: the next element of the list is added to the accumulated result *before* the recursive call. Because of this, there are no deferred operations, thus it is tail-recursive. (Recall that the official definition of *tail recursive* is: no deferred operations on any recursive call.) The initial value of the accumulated result that is passed to the auxiliary function `sum'` on the first call is the basis element `0`. Thus the elements are added in the opposite order

$$(((\ldots((0 + x_1) + x_2) + \ldots) + x_{n-1}) + x_n),$$

from left to right. Luckily in this case, addition is associative, so the order of additions doesn't matter, and we will get the same result. We would not be so fortunate if we tried to do the same thing with a nonassociative operation such as exponentiation or `::`.

The third version gives us the best of both: performance of the operations from right to left, but with no deferred operations. This is done using **continuations**. Since we have first-class functions, we can create a

function that packages up any such deferred operations, then passes it down in the recursive call for somebody else to perform. In this example, the auxiliary function `sum'` takes a continuation function as an extra argument, which consists of all the deferred operations accumulated so far. Before the next recursive call, the deferred addition is composed with the previously accumulated ones in the correct order, and this new continuation function is passed down in the next recursive call. At the bottom of the recursion, all the deferred operations that were accumulated on the way down are performed all at once by applying the continuation to the basis element `0`.

The initial call to `sum'` passes the identity function `fn x => x`, which is the identity element for the operation of function composition. On the way down, the operation of adding the new element `x` is composed with the passed-in continuation `k` consisting of all the deferred additions accumulated so far, giving a new continuation `fn a => k (x + a)`. Note that this is equivalent to `k o (fn a => x + a)`.

Thus at the bottom of the recursion on input $[x_1,x_2,...,x_n]$, we have a function that is equivalent to the composition

$$(\text{fn x -> x}) \circ (\text{fn a -> } x_1 + a) \circ (\text{fn a -> } x_2 + a) \circ \ldots \circ (\text{fn a -> } x_{n-1} + a) \circ (\text{fn a -> } x_n + a)$$

where `o` denotes function composition (i.e. `f o g` is `f(g(x))`).

Applying this function to 0 gives

$$(x_1 + (x_2 + (\ldots + (x_{n-1} + (x_n + 0))\ldots))),$$

which is the same sum calculated in the same order as in the first version.

Note that even though our third version is tail recursive, it does not really save any space over the first version as the second one does. Both the first and third versions must remember the deferred operations, but they do so in different places: the first on the runtime stack, the third in the closures of continuations. The two representations require asymptotically the same amount of space to within a constant factor. So to say that tail recursion saves stack space, while strictly true, is not the whole story.

Let's see what happens when we give `sum` an infinitely long list.

```
let rec inf = 0::inf in
    sum inf
```

The first (non tail-recursive) version causes a stack overflow. The second (tail-recursive) version runs forver and uses a constant amount of memory. The third (using continuations) version uses memory proportional to the number of items it has processed, so it will run until there is no more memory left.

Two versions of `fold_right`

The construction above has nothing to do with addition. It is completely general and would work for any operation, associative or not. What better way to illustrate this than to do it for `fold_right`, which applies an *arbitrary* operation from right-to-left over a list, starting from an *arbitrary* basis element. Here are the ordinary recursive definitions of `fold_right` and a tail-recursive version using continuations, corresponding to the first and third `sum` programs above. One can try to formulate a tail-recursive version corresponding to the second `sum` program above, but it is not clear how to do this without reversing the list or resorting to other machinations.

```
(* straightforward recursion *)
let rec fold_right (f : 'a -> 'b -> 'b) (s : 'a list) (b : 'b) : 'b =
  match s with
      [] -> b
    | x::xs -> f x (fold_right f xs b)

(* using continuations *)
let fold_right (f : 'a -> 'b -> 'b) (s : 'a list) (b : 'b) : 'b =
  let rec fold_right' s k =
    match s with
        [] -> k b
      | x::xs -> fold_right' xs (fun y -> k (f x y)) in
    fold_right' s (fun x -> x)
```

Let's prove these two functions are equivalent. The first version is

```
let rec fold_right f s b =
  match s with
      [] -> b
    | x::xs -> f x (fold_right f xs b)
```

and the auxiliary function in the second (for fixed `f` and `b`) is

```
let rec fold_right' s k =
    match s with
        [] -> k b
      | x::xs -> fold_right' xs (fun y -> k (f x y))
```

We will prove by induction on `s` that for any `k`,

```
fold_right' s k  =  k (fold_right f s b).
```

Basis: `s = [].`

```
fold_right' [] k  =  k b  =  k (fold_right f [] b)
```

by the first clauses in the definitions of `fold_right'` and `fold_right`.

Induction step: `s = x::xs.`

```
fold_right' (x::xs) k  =  fold_right' xs (fun y ->  k (f x y))
                        by the second clause in the definition of fold_right'
                   =  (fun y -> k (f x y)) (fold_right f xs b)
                        by the induction hypothesis
                   =  k (f x (fold_right f xs b))
                        by the substitution model
                   =  k (fold_right f (x::xs) b)
                        by the second clause in the definition of fold_right.  QED
```

In particular, for `k` the identity function `fn x => x` on which `fold_right'` is initially called,

```
fold_right' s (fn x -> x)  =  fold_right f s b
```

thus the two versions of `fold_right` above are equivalent.

First-Class Continuations in SML/NJ

Unfortunately, OCaml does not have first-class continuations (it is difficult to implement them efficiently), but a very similar language called SML/NJ does have them. We will give code examples in SML/NJ, but you shouldn't have much trouble following along. One problem with transforming a program from ordinary recursion to continuation-passing style is that we must sometimes do quite a bit of work to figure out how to package up the continuation in the correct way. SML/NJ provides a nice facility for doing this automatically. There is a function `callcc` ("call with current continuation") that figures out what the correct continuation would be and automatically passes it in. This seems a little magical, because the continuation is not explicitly represented. But it has the same effect as the function that we would have constructed ourselves to perform the deferred computations. The nice thing about this is that it allows us to write the function in ordinary recursive style with deferred operations, but then do a very simple modification involving `callcc` to make it tail recursive in continuation-passing style.

In general, a **continuation** in the SML/NJ inplementation of `callcc` is a *dynamic* object (created at runtime, not determined by the program text but by the computation history) that is created when a certain expression `e` is about to be evaluated. The continuation consists of a function that packages up in its closure the state of the computation that will be in effect when `e` returns. Later on, the computation can be restarted in that state just by calling the function. When the function is called with input `v`, computation will begin in the saved state exactly as if we had just returned from the evaluation of `e` and it had returned the value `v`. To call the function associated

with a continuation `c`, we evaluate `throw c v`. You cannot call continuations created with `callcc` as ordinary functions, you must invoke them with `throw`. These functions are defined in the `SMLofNJ.Cont` structure.

Let's illustrate the use of `callcc` and `throw` with some examples.

The 3x+1 Problem

The `3x+1` problem refers to the following iteration. Given a positive integer `x`,

- if `x` is even, divide it by `2`
- if `x` is odd, multiply it by `3` and add `1`.
- repeat forever.

This process eventually ends up in the cycle `1 -> 4 -> 2 -> 1` for all `x` that anyone has every tried, but no one has yet been able to prove that this occurs for all `x`.

Here are three programs that perform this iteration, returning the sequence of integers produced along the way until `1` is encountered. These three programs correspond to the three versions of `sum` above: ordinary recursion, tail recursion, and continuation-passing style.

```
fun even x = x mod 2 = 0

fun txp1 x =
  if x = 1 then [1]
  else if even x then x::txp1(x div 2)
  else x::txp1(3*x + 1)

fun txp2(x,a) =
  if x = 1 then a
  else if even x then txp2(x div 2,a@[x])
  else txp2(3*x + 1,a@[x])

fun txp3(x,k) =
  if x = 1 then k [1]
  else if even x then txp3(x div 2,fn a => k(x::a))
  else txp3(3*x + 1,fn a => k(x::a))


- txp1 100;
val it = [100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1] : int list
- txp2 (100,[]);
val it = [100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1] : int list
- txp3 (100,fn x => x);
val it = [100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1] : int list
```

Note that `txp1` and `txp3` can use `::`, a constant-time operation, whereas the tail recursive version `txp2` must use the linear-time `@`. This is because of the associativity issues discussed above. We could fix this by producing the list in the opposite order and then reversing it at the end, but this is not the point.

Using `callcc`, we can convert `txp1` to tail-recursive form without a whole lot of rewriting. The general procedure for converting an ordinary recursive function `foo` to a continuation-passing version `foo'` consists of the following steps:

1. modify `foo` to take an extra argument, which will be a continuation. The new function `foo'` should be curried so that the continuation is its last argument.
2. Replace every call `foo(e)` with `callcc(foo'(e))`. When evaluated, this will create an invisible continuation and pass it as the last argument to `foo'(e)` (which, as you recall from step 1, now takes a continuation as its last argument).
3. for the base cases, instead of returning a value `v` directly, throw it to the passed-in continuation `k` by evaluating `throw k v`.

Applying this to `txp1`, we obtain

```
val callcc = SMLofNJ.Cont.callcc
val throw = SMLofNJ.Cont.throw

fun txpcc x k =
  if x = 1 then throw k [1]
  else if even x then x::callcc(txpcc(x div 2))
  else x::callcc(txpcc(3*x + 1))

- callcc (txpcc 100);
val it = [100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,
10,5,16,8,4,2,1] : int list
```

Here is another example involving the Fibonacci numbers.

Before:

```
fun fib n =
  if n <= 1 then n
  else fib(n-1) + fib(n-2)
```

```
- fib 10;
val it = 55 : int
```

After:

```
fun fibcc n k =
  if n <= 1 then throw k n
  else callcc(fibcc(n-1)) + callcc(fibcc(n-2))
```

```
- callcc(fibcc 10);
val it = 55 : int
```

Examples using Continuations

Suspend/Resume

Using continuations, we can jump out of the middle of a computation and later jump back in. Consider the following simple program that goes down into a recursion and comes back up, printing messages along the way.

```
fun println s = print (s^"\n")

fun f x =
  if x = 0 then ()
  else (println (Int.toString x ^ " down");
        f (x-1);
        println (Int.toString x ^ " up"))

- f 5;
5 down
4 down
3 down
2 down
1 down
1 up
2 up
3 up
4 up
5 up
val it = () : unit
```

Now we can modify this to jump out at the bottom of the recursion, setting a global continuation so that we can later jump back and resume the computation where we left off.

```
(* global continuations, initialized to dummy values *)
val dummy = SMLofNJ.Cont.isolate (fn() => ())
val esc = ref dummy
val sav = ref dummy
val bye = ref dummy

fun f x =
  if x = 0 then callcc (fn c => (sav := c; throw (!esc) ()))
  else (println (Int.toString x ^ " down");
        f (x-1);
        println (Int.toString x ^ " up"))

fun down x = (callcc (fn c => (esc := c; f x; throw (!bye) ())))
fun up () : unit = callcc (fn d => (bye := d; throw (!sav) ()))

- down 5;
5 down
4 down
3 down
2 down
1 down
val it = () : unit    <- here we have jumped out
- 299 + 13;           <- now we do some arbitrary stuff
val it = 312 : int
- up ();              <- and jump back in to resume where we left off
1 up
2 up
3 up
4 up
5 up
val it = () : unit
- up ();              <- we can even jump back in multiple times
1 up
2 up
3 up
4 up
5 up
val it = () : unit
```

Expanding on this idea, we can give a general suspend/resume facility that essentially allows us to set breakpoints, jumping out of the program and into the interpreter, then later resuming where we left off.

```
val sus = ref dummy
val res = ref dummy

fun try f x = callcc (fn c => (sus := c; f x; throw (!sus) ()))
fun resume () : unit = callcc (fn d => (sus := d; throw (!res) ()))
fun suspend msg : unit = callcc (fn d => (res := d; println msg; throw (!sus) ()))

fun fact n =
 (suspend ("fact " ^ Int.toString n);
  if n <= 0 then 1 else n * fact(n-1))


- try fact 4;
fact 4
```

```
val it = () : unit
- resume();
fact 3
val it = () : unit
- resume();
fact 2
val it = () : unit
- resume();
fact 1
val it = () : unit
- resume();
fact 0
val it = () : unit
- resume();
val it = () : unit
```

Simulating Gotos

Here is an ugly little piece of code that is not even legal ML (no such thing as labels and goto).

```
fun product (lst:int list ref) : unit = let
  val p = ref 1
in
  LOOP:
    if !lst = [] then goto OUT
    p := !p * hd (!lst);
    lst := tl (!lst);
    if !p <> 0 then goto LOOP;
  OUT:
    print ("Product=" ^ Int.toString (!p))
end
```

We can simulate the labels with global continuations and the gotos with throws (not that you would ever want to do this).

```
val LOOP = ref dummy
val OUT = ref dummy

fun goto c = throw (!c) ()

fun product (lst:int list ref) : unit = let
  val p = ref 1
in
  callcc (fn c => (OUT := c;
    callcc (fn c => LOOP := c);
    if !lst = [] then goto OUT else ();
    p := !p * hd (!lst);
    lst := tl (!lst);
    if !p <> 0 then goto LOOP else ()));
  println ("Product=" ^ Int.toString (!p))
end

- product (ref [3,8,13]);
Product=312
val it = () : unit
```

In fact, truth be told, continuations are really just glorified gotos.

Nonlocal Control Flow

Here is an extended example illustrating the use of continuations to achieve nonlocal flow of control. This is useful in interactive systems in which we want to respond to errors by rolling back to some primal state. The difference between this and just calling an error routine is that if the error occurs very deep in some recursive computation, the partial results that were on the stack at the time of the error will still be there. By setting an error handler and calling it with a global continuation, we can transfer control to the error handler from anywhere in the program and restart the computation afresh with a clean stack.

```
val exitCont = ref dummy  (* global exit continuation *)
val errorCont = ref dummy (* global error continuation *)

fun echo args = app println args
fun echo2 args = (echo args; echo args)
fun reverse args = app (println o implode o rev o explode) args
fun error args = (print "uh-oh! "; throw (!errorCont) ())
fun quit args = throw (!exitCont) ()

val commands =
  [("echo",echo),
   ("echo2",echo2),
   ("reverse",reverse),
   ("error",error),
   ("quit",quit)]

fun help() =
 (println "Available commands are:"; app println (map #1 commands))

(* read-eval-print loop *)
fun repl() = let
  val inputLine = (print ">"; TextIO.inputLine TextIO.stdIn)
  val tokenizedInput = String.tokens Char.isSpace inputLine
  fun process(cmd,arg) =
    case (List.find (fn(c,_) => String.isPrefix cmd c) commands) of
      SOME (_,action) => action arg
    | NONE => help()
in
  case tokenizedInput of
    cmd::args => process(cmd,args)
  | [] => help();
  repl()
end

fun setErrorHandler ret =
 (callcc (fn d => (errorCont := d; throw ret ()));
  println "something bad happened!";
  help();
  repl())

fun go() = callcc (fn c =>

 (SMLofNJ.Internals.GC.messages false;
  exitCont := c;
  callcc setErrorHandler;
  repl()))


- go();
>echo2 hello world
hello
world
```

```
hello
world
>rev hello world
olleh
dlrow
>error
uh-oh! something bad happened!
Available commands are:
echo
echo2
reverse
error
quit
>q
val it = () : unit
```