Continuation Passing Style Lab

`cps_lab`

Principles of Functional Programming

Summer 2023

Contents

1 Continuations

1.1 Definitions

Recall: A function is **tail-recursive** if the function calls itself recursively and does no further computation (i.e. directly returns the result of the recursive call).

A **continuation** is an additional argument to a function that describes “what’s left to do” in order to complete a given computation. Continuations achieve this by being a function.

A function, `f`, is written in **continuation passing style** (CPS) if:

1. `f` takes at least one continuation as an argument
2. If `f` makes a call to a recursive function `g`, then this call is a tail call and `g` must be in CPS.
3. If `f` makes a call to a function `g`, which itself has continuations, then this call is a tail call and `g` must be in CPS.
4. `f` calls its continuation(s) and does so in tail call(s)

1.2 Thinking with Continuations

Task 1.1.

Let’s look at the `fact` function, which computes the factorial of a non-negative integer:

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

How would we write this in CPS? And what would be the type of that function `fact'`?

```
(* fact' :
 * REQUIRES:
 * ENSURES:
 *)
```

Task 1.2.

Let’s trace out a call to `fact'`.

```
fact' 3 Int.toString
```

1.3 Uses of Continuations

The two main uses of CPS are *accumulation* and *control flow*. As we've seen with factorial, CPS can help us accumulate a result.

The second use, control flow, is achieved by encoding "what to do next" in the continuation functions.

Task 1.3.

Let's say we wanted to search for an element in a tree, given some predicate. We want to find the first element that satisfies this predicate (if it exists).

```
(* search : ('a -> bool) -> 'a tree -> 'a option *)
fun search p Empty = NONE
  | search p (Node (L, x, R)) =
    if p x then SOME x else
      (case search p L of
           SOME y => SOME y
         | NONE   => search p R)
```

How can we write this function in CPS? (Let's begin with the type.)

```
(* search' :
 * REQUIRES:
 * ENSURES:


 *)
```

1.4 CPS Writing Tips

Here are some tips for writing functions in CPS:

- **In the base case**: Call the continuation on the base case return value instead of directly returning the value.
- **In the recursive case**:
  - The continuation acts either as a functional accumulator or as a way of directing control flow.
  - Generally, the first thing you want to do is make the recursive call (since you know the function must be tail recursive) and modify the continuations to that recursive call.
  - **When in doubt, start with the recursive call!**
- **In general**:
  - If you find yourself getting type-mismatch errors, you are likely trying to case or modify the result of a CPS call (we cannot case on the result of a recursive call since CPS functions return a polymorphic type, which we know nothing about). Make sure all your function calls are tail-calls.
  - If you are are getting confusing type errors, you might be able to get more specific errors if you add type annotations to your function.
  - You may only use helper functions if they are non-recursive, are CPS functions, or are explicitly allowed by the problem statement
  - `fn x => k x` $\cong$ `k`, so if you find yourself writing something that looks like `fn x => k x`, you can replace it with just `k`.

2 PS: Can you C if this is CPS?

We discussed how CPS is a way of passing functions in as arguments and using the continuations as functional accumulators or as a way to direct control flow.

Here are a few good guidelines to follow:

- In the base cases, instead of directly returning a value, we apply the continuation to the result
- In the recursive cases, the continuation acts as a functional accumulator and/or a way of directing control flow
- All calls to functions with continuations must be tail calls

You **may** do the following in a CPS function:

- Case on the value of the input
- Use `let..in..end` expressions
- Use non-recursive helper functions in non-tail calls
- Use or case on the result of predicates in non-tail calls
- Pass in modified continuations, predicates, and/or other arguments to the recursive calls you make

You **may not** do the following in a CPS function:

- Manipulate, case on, or otherwise use the result of either recursive calls, CPS helper functions, or continuation calls. (This would break the tail-call requirement.)
- Use a non-CPS recursive helper function (unless explicitly permitted by the question).

For a more rigorous definition of CPS and more examples of functions that are and are not CPS, please refer to the appendix in the CPS HW.

That being said, you can now figure out whether the following functions are written in CPS.

Task 2.1. (Recommended)

```
fun isCps1 (0 : int) (k : int -> 'a) : 'a  = k 0
  | isCps1 n k = isCps1 (n-1) (fn res => k (n + res))
```

Task 2.2. (Recommended)

```
fun isCps2 (x : int) (k : int -> 'a) : 'a =
  if x < 0 then k 0 else
    (case (isCps2 x (fn y => y)) of
        0 => k 0
       | v => k (v - 1))
```

Task 2.3. (Recommended)

```
fun helper ([] : 'a list) : int = 0
 |  helper (x::xs) = 1 + helper xs

fun isCps3 (L : 'a list) (k : int -> 'b) : 'b = k (helper L)
```

Task 2.4. (Recommended)

```
fun isCps4 ([] : string list) (k : string list -> 'a) : 'a = k []
 |  isCps4 (x::xs) k = isCps4 xs
                         (fn res => k (("haha" ^ x) :: res))
```

Task 2.5. (Recommended)

```
fun isCps5 ([] : int list) (p : int -> bool) (k : bool -> 'a) : 'a =
      k (p 0)
  | isCps5 (x::xs) p k = isCps5 xs (fn y => p (x + y)) k
```

3 Can we continue?

Any recursive function can actually be written using CPS!

For practice, you will write the following recursive functions using CPS.

Task 3.1. (Recommended)

Recall the function `length : 'a list -> int` which calculates the length of a list:

```
fun length [] = 0
  | length (x::xs) = 1 + length xs
```

Write the CPS version:

`length' : 'a list -> (int -> 'b) -> 'b`

REQUIRES: true

ENSURES: `length' L k` ≅ `k (length L)`

Let the ENSURES clause guide you!

Task 3.2.

Prove your implementation of `length'` correct.

That is, prove that, for all types `t1, t2`, all values `L : t1 list` and all function values `k : int -> t2`, `length' L k` ≅ `k (length L)`.

You may cite the totality of `length` without proof.

Task 3.3. (Recommended)

Recall the function `size : 'a tree -> int` which takes a tree and returns the number of nodes in the tree:

```
fun size Empty = 0
  | size (Node (L, x, R)) = 1 + size L + size R
```

Write the CPS version:

`size' : 'a tree -> (int -> 'b) -> 'b`

REQUIRES: true

ENSURES: `size' T k` ≅ `k (size T)`

Let the ENSURES clause guide you!
Also, try including a recursive call *inside the continuation of another recursive call.*

Task 3.4.

Recall the function `filter : ('a -> bool) -> 'a list -> int` which takes a predicate `p` and a list `L` and returns a list containing only those elements `x` of `L` such that `p x` $\Longrightarrow$ `true`:

```
fun filter p [] = []
  | filter p (x::xs) =
      (case p x of
          true => x :: (filter p xs)
        | false => filter p xs)
```

Write the CPS version:

`filter' : ('a -> bool) -> 'a list -> ('a list -> 'b) -> 'b`

REQUIRES: `p x` is valuable for all `x` in `L`

ENSURES: `filter' p L k` $\cong$ `k (filter p L)`

Let the ENSURES clause guide you!

Task 3.5.

Recall the function `@` which puts one list after another:

```
fun [] @ B = B
  | (x::xs) @ B = x::(xs @ B)
```

Write the CPS version:

`append' : ('a list * 'a list) -> ('a list -> 'b) -> 'b`

REQUIRES: true

ENSURES: `append' (A, B) k` $\cong$ `k (A @ B)`

Task 3.6.

Recall the function `rev : 'a list -> 'a list` which reverses a list:

```
fun rev [] = []
  | rev (x::xs) = (rev xs) @ [x]
```

Write the CPS version:

`rev' : 'a list -> ('a list -> 'b) -> 'b`

REQUIRES: true

ENSURES: `rev' L k` $\cong$ `k (rev L)`

You may want to use `append'` from the previous task.

4 Search & Continue

Recall the CPS function `search'`, which searches through a tree recursively for an element that satisfies the predicate `p`:

```
fun search' p Empty sc fc = fc ()
  | search' p (Node(L,x,R)) sc fc =
      if p x then sc x
      else search' p L sc (fn () => search' p R sc fc)
```

This code is tail recursive, clean, and allows us to customize both a success continuation `sc`, which will be applied to the result of our search, and a failure continuation `fc`, which will be called if no predicate-satisfying element is found.

4.1 Direction Find

Suppose we were interested in *where* in the tree the element was located.

Above, we passed `x` into `sc`, where `x` satisfies our predicate `p`. Now, we want our `sc` to take in a `direction list`, which tells how to traverse the tree from the root to the found `x`.

To do so, we make the following datatype definition:

```
datatype direction = Left | Right
```

A `direction list` tells us how to traverse the tree:

- peel off the head of the list,
- if it's `Left` then go into the left subtree,
- if it's `Right` then go into the right subtree.
- Stop when the list is empty.

This is encoded by the function `traverse : 'a tree -> direction list -> 'a`:

```
exception NotFound

fun traverse Empty _ = raise NotFound
  | traverse (Node (L, x, R)) [] = x
  | traverse (Node (L, x, R)) (Left::xs) = traverse L xs
  | traverse (Node (L, x, R)) (Right::xs) = traverse R xs
```

Now onto your task.

Task 4.1.

In continuation passing style, write a function

```
directionFind :
                ('a -> bool)             (* predicate function *)
                -> 'a tree               (* the tree to look through *)
                -> (direction list -> 'b) (* success continuation *)
                -> (unit -> 'b)          (* failure continuation *)
                -> 'b                    (* result *)
```

such that:

$$\texttt{directionFind p T sc fc} \Longrightarrow \begin{cases} \texttt{sc dir} & \text{such that } \texttt{p (traverse T dir)} \Longrightarrow \texttt{true} \\ \texttt{fc ()} & \text{if no such } \texttt{dir} \text{ exists} \end{cases}$$

Here's some examples:

```
val success = SOME
val failure = fn () => NONE
val p x = x > 0

val T0 = Empty
val T1 = Node(Empty,1,Empty)
val T2 = Node(T1,~3,T1)
val T3 = Node(Empty,0,Node(T1,0,Empty))
```

```
directionFind p T0 success failure ⟹ NONE
directionFind p T1 success failure ⟹ SOME []
directionFind p T2 success failure ⟹ SOME [Left]
directionFind p T3 success failure ⟹ SOME [Right,Left]
```

5 Continuations continued

Recall the shrub datatype

```
datatype 'a shrub = Leaf of 'a | Branch of 'a shrub * 'a shrub
```

Shrubs, although similar to trees, store values at the *leaves* instead of the *nodes*. Additionally, there's no empty shrub.

Let's implement some shrub functions, shall we?

Task 5.1.

Write the CPS function

```
mult : int shrub -> (int -> 'a) -> 'a
```

such that `mult T k` evaluates to `k v`, where `v` is the product of the elements in `T`.

It is often the case that we will not need to inspect the entire shrub.

For example, if we encounter `Leaf 0`, we can assume that the entire product will return zero. In the following problem you will use continuations to write a function to short circuit immediately after seeing the first zero.

Task 5.2.

Write a CPS function

```
mult' : int shrub -> (int -> 'a) -> (unit -> 'a) -> 'a
```

such that

$$\texttt{mult' T sc fc} \Longrightarrow \begin{cases} \texttt{sc v} & \text{if all the elements of } \texttt{T} \text{ are nonzero and their product is } \texttt{v} \\ \texttt{fc ()} & \text{if } \texttt{T} \text{ contains a zero} \end{cases}$$

Constraint: `mult' T sc fc` should never do any multiplications in which either integer is zero.

Task 5.3.

Using only `mult'`, write the function

```
divShrub : int * int shrub -> int option
```

which takes an integer `x` and a shrub `T`, and returns `SOME (x div v)` where `v` is the product of the elements of `T`, or `NONE` if `T` contains a zero.

Constraint: Your implementation of `divShrub` should never perform any multiplication in which either integer is zero.

Constraint: `divShrub` itself should not case on the structure of `T` (all recursion should happen inside `mult'`).