## Contents

1 Continuations
  1.1 Definitions ................................................. 2
    Task 1. .................................................... 2
    Task 2. .................................................... 2
  1.2 Thinking with Continuations ................................. 2
    Task 3. .................................................... 2
    Task 4. .................................................... 3
  1.3 Uses of Continuations ...................................... 3
    Task 5. .................................................... 3

2 PS: Can you C if this is CPS? ................................. 4
  Task 6. ..................................................... 4
  Task 7. ..................................................... 4
  Task 8. ..................................................... 4
  Task 9. ..................................................... 4
  Task 10. ..................................................... 5

3 Can we continue? ................................................ 6
  Task 11. ..................................................... 6
  Task 12. ..................................................... 6
  Task 13. ..................................................... 6
  Task 14. ..................................................... 7
  Task 15. ..................................................... 7

4 Search & Continue .............................................. 8
  4.1 Direction Find ............................................ 8
    Task 16. .................................................. 8

5 Continuations continued ...................................... 10
  Task 17. ..................................................... 10
  Task 18. ..................................................... 10
  Task 19. ..................................................... 10
1 Continuations

1.1 Definitions

A continuation is an additional argument to a function describing “what’s left to do” in order to complete a given computation. We achieve this by letting the argument be a function.

Task 1.

A function is written in continuation passing style (CPS) if:

1.
2.
3.

Task 2.

Definition of tail-recursive:

1.2 Thinking with Continuations

Task 3.

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

```plaintext
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 factCPS?

(* factCPS : *)
Task 4.

Let’s trace out a call to `factCPS`.

```
factCPS 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 comes from this idea of keeping “what to do next” as a function to represent control flow.

Task 5.

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 any exist).

```
(* 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?

```
(* search' : *)
```
2 PS: Can you C if this is CPS?

What is CPS? While a seemingly difficult concept to grasp, we hope that this lab will give you a more structured understanding of how to write code in CPS.

Remember what our definition for CPS was in lecture. 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 base cases, we apply the continuation instead of directly returning a value
- In recursive cases, the continuation acts as a functional accumulator or as a way of directing control flow
- All calls to CPS functions 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
- Write recursive functions

You **may not** do the following:

- Manipulate, case on, or otherwise use the result of recursive calls as this would break the tail-recursive requirement for CPS functions

With that being said, you can figure out whether the following functions are written in CPS or not.

**Task 6.**

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

**Task 7.**

```ml
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 8.**

```ml
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 9.**
fun addHaha s = s ^ "Haha"

fun isCps4 (_: string list) addHaha (k: string list -> 'a) : 'a = 
k []
| isCps4 (x::xs) addHaha k = isCps4 xs addHaha
    (fn res => k ((addHaha x)::res ))

Task 10.

fun isCps5 (f: int -> ('a -> 'a) -> int) (x: int) (k: int -> 'b) : 'b =
case (f x (fn y => y)) of
    0 => k 1
| n => k (n - 1)
3 Can we continue?

Any normal recursive function can be written in continuation passing style. For practice, you will write the following recursive functions in continuation passing style.

Task 11.
Recall the function `length : 'a list -> int` which calculates the length of a list:

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

Write a CPS function

```ml
length' : 'a list -> (int -> 'b) -> 'b
REQUIRES: true
ENSURES: length' L k ≃ k(length L)
```

Let the ENSURES clause guide you!

Task 12.
Recall the function `size : 'a tree -> int` which takes a tree and returns the number of nodes in the tree:

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

Write a CPS function

```ml
size' : 'a tree -> (int -> 'b) -> 'b
REQUIRES: true
ENSURES: size' T k ≃ k(size T)
```

Let the ENSURES clause guide you! *Hint: Try including a recursive call inside the continuation of another recursive call.*

Task 13.
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) ==> true`:

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

Write a CPS function
Let the ENSURES clause guide you!

**Task 14.**
Prove your implementation of \texttt{length'} correct. That is, prove that for all types \(t_1, t_2\), all values \(L : t_1\ list\) and all functions \(k : \texttt{int} \rightarrow t_2\) that \(\texttt{length'} L k \equiv k(\texttt{length} L)\).

**Task 15.**
Recall the function \texttt{rev}' : \(\texttt{a list} \rightarrow \texttt{a list}\) which reverses a list:

\[
\begin{align*}
\texttt{fun rev} \ [\] &= [\] \\
\mid \texttt{rev} \ (x::xs) &= (\texttt{rev} \ xs) @ [x]
\end{align*}
\]

Write a CPS function

\[
\begin{align*}
\texttt{rev'} : \texttt{a list} \rightarrow (\texttt{a list} \rightarrow \texttt{b}) \rightarrow \texttt{b} \\
\text{REQUIRES:} & \quad \texttt{true} \\
\text{ENSURES:} & \quad \texttt{rev'} L k \equiv k(\texttt{rev} L)
\end{align*}
\]

Let the ENSURES clause guide you!
4 Search & Continue

Consider the following function which searches through a tree recursively for an element that satisfies the predicate \( p \):

\[
\begin{align*}
\text{fun treeFind } p \text{ Empty} &= \text{false} \\
| \text{treeFind } p \text{ (Node(L,x,R))} &= (p \ x) \ orelse \ (\text{treeFind } p \text{ L}) \ orelse \ (\text{treeFind } p \text{ R})
\end{align*}
\]

But this is kinda boring because it tells us whether or not there’s an element in the tree which satisfies our predicate, but if so it doesn’t tell us what that element is.

We can remedy this by generalizing our function to return an option instead of a boolean:

\[
\begin{align*}
\text{fun treeFindOpt } p \text{ Empty} &= \text{NONE} \\
| \text{treeFindOpt } p \text{ (Node(L,x,R))} &= \text{if } p(x) \text{ then } \text{SOME}(x) \text{ else } \\
& \text{case } (\text{treeFindOpt } p \text{ L}, \text{treeFindOpt } p \text{ R}) \text{ of } \\
& (\text{SOME } v, _) \Rightarrow \text{SOME } v \\
& (_, \text{SOME } v) \Rightarrow \text{SOME } v \\
& (_, \text{NONE}, \text{NONE}) \Rightarrow \text{NONE}
\end{align*}
\]

But this contains a lot of ugly casing. It turns out that CPS allows us to solve this:

\[
\begin{align*}
\text{fun treeFindCPS } p \text{ Empty } sc \ fc &= fc () \\
| \text{treeFindCPS } p \text{ (Node(L,x,R)) } sc \ fc &= \text{if } p(x) \text{ then } sc \ x \text{ else } \\
& \text{treeFindCPS } p \text{ L } sc \ (\text{fn } () \Rightarrow \text{treeFindCPS } p \text{ R } sc \ fc)
\end{align*}
\]

This code is tail recursive, clean, and allows us to supply whatever continuation \( sc \) we want to apply to the result of our search and whatever continuation \( fc \) we want to call if nothing in the tree satisfies the predicate.

4.1 Direction Find

Suppose we were interested in where in the tree the element was located. What we want to do is have our success continuation take in a set of directions which tells us where to look in the tree for an element that satisfies the predicate. To do so, we make the following datatype definition:

\[\text{datatype direction} = \text{Left} | \text{Right}\]

A list of values of this type 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:

\[
\begin{align*}
\text{exception NotFound} \\
\text{fun traverse } \text{Empty} _ &= \text{raise } \text{NotFound} \\
| \text{traverse } \text{(Node(L,x,R)) } [] &= x \\
| \text{traverse } \text{(Node(L,x,R)) } (\text{Left::xs}) &= \text{traverse } \text{L } xs \\
| \text{traverse } \text{(Node(L,x,R)) } (\text{Right::xs}) &= \text{traverse } \text{R } xs
\end{align*}
\]

Now onto your task.

Task 16.
Write a function

\[
\text{treeFindCPS}' : \quad \langle \text{'a -> bool} \rangle \quad \langle \text{the tree to look through} \rangle \\
\quad \langle \text{direction list -> 'a} \rangle \quad \langle \text{success continuation} \rangle \\
\quad \langle \text{unit -> 'a} \rangle \quad \langle \text{failure continuation} \rangle \\
\quad \langle \text{'a} \rangle 
\]

such that:

\[
\text{treeFindCPS}' \ p \ T \ sc \ fc \ \Rightarrow \begin{cases} 
\text{sc dir such that p(traverse T dir) \Rightarrow true} \\
\text{fc () if no such dir exists}
\end{cases}
\]

Here’s some examples:

```plaintext
val success = SOME
val failure = fn () => NONE
fun 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))

val NONE = treeFindCPS' p T0 success failure
val SOME [] = treeFindCPS' p T1 success failure
val SOME [Left] = treeFindCPS' p T2 success failure
val SOME [Right,Left] = treeFindCPS' p T3 success failure
```
5 Continuations continued

Recall the shrub datatype

\[
\text{datatype} \quad \text{'a shrub} = \text{Leaf of 'a | Branch of 'a shrub * 'a shrub}
\]

Shrubs are similar to trees, except that they store values at the leaves instead of the nodes, and that there’s no empty shrub. In this lab (and on the homework) we’ll have you implement several functions on shrubs.

Task 17.

Write a CPS function \( \text{mult} : \text{int shrub} \rightarrow (\text{int} \rightarrow \text{'a}) \rightarrow \text{'a} \) such that \( \text{mult} \ T \ k \) evaluates to \( k(v) \), where \( v \) is the product of the elements in \( T \).

Task 18.

It is often the case that we will not need to look at the entire shrub. For example, if we see \( \text{Leaf} \ 0 \), 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.

Write a CPS function \( \text{mult'} : \text{int shrub} \rightarrow (\text{int} \rightarrow \text{'a}) \rightarrow (\text{unit} \rightarrow \text{'a}) \rightarrow \text{'a} \) such that \( \text{mult'} \ T \ sc \ fc \) should never do any multiplications in which either integer is zero.

Task 19.

Using only \( \text{mult'} \), write a function \( \text{divShrub} : \text{int} * \text{int shrub} \rightarrow \text{int option} \) which takes an integer \( x \) and a shrub \( T \) and returns \( \text{SOME}(x \ \text{div} \ v) \) where \( v \) is the product of the elements of \( T \), or \( \text{NONE} \) if \( T \) contains a zero.

Your implementation of \( \text{divShrub} \) should never perform any multiplication in which either integer is zero, \( \text{divShrub} \) itself should not case on the structure of \( T \) (all recursion should happen inside \( \text{mult'} \)).