Learning Objectives

1. Explain the characteristics of continuations and functions written in continuation passing style (CPS).

2. Critique code that represents a failed attempt to convert a function to CPS.

3. Convert some functions to CPS using the supplied transformation rules.

4. Change the result of a computation by reordering continuations.

Characteristics

<table>
<thead>
<tr>
<th>Direct Style</th>
<th>CPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  fact 0 = 1</td>
<td>0  fact 0 k = k 1</td>
</tr>
<tr>
<td>1  fact n = n * fact (n-1)</td>
<td>1  fact n k = fact (n-1)(v -&gt; k (n * v))</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3  sumList [] = 0</td>
<td>3  sumList [] k = k 0</td>
</tr>
<tr>
<td>4  sumList (x:xs) = x + sumList xs</td>
<td>4  sumList (x:xs) k = sumList xs (v -&gt; k (x + v))</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6  prodList [] = 1</td>
<td>6  prodList xx k = aux xx k k</td>
</tr>
<tr>
<td>7  prodList (0:xs) = 0</td>
<td>7  where aux [] ks ka = ks 1</td>
</tr>
<tr>
<td>8  prodList (x:xs) = x * prodList xs</td>
<td>8  aux (0:xs) ks ka = ka 0</td>
</tr>
<tr>
<td></td>
<td>9  aux (x:xs) ks ka = aux xs (v -&gt; ks (x * v)) ka</td>
</tr>
</tbody>
</table>

Problem 1) What is the relationship between the return value of a direct-style function and the value passed into k in the CPS equivalent functions?

Problem 2) There are several anonymous functions in the CPS code. Their parameters are all named v. What kinds of values are being passed into these parameters?
**Problem 3)** What percentage of the recursive calls are in tail form in the CPS functions?

**Problem 4)** We like to pretend that functions written in CPS "never return". What justifies that?

**Problem 5)** Suppose we call direct style prodList with argument \([2,3,0,4,9,8,7]\). How many times will the multiplication operator be invoked? Suppose we call CPS prodList with the same list and continuation print. (The print function prints its argument to the screen and does not return any value\(^1\).)

---

**Now It’s Ruined**

Consider the following three programs. They are not in CPS.

```
0 decList [] k = []
1 decList (x:xs) k = (x-1 : decList xs k)
```

```
0 maxList [] k = k
1 maxList [x] k = k x
2 maxList (x:xs) k = maxk x (maxList xs k) k
```

```
0 minList [] k = k x
1 minList [x] k = k x
2 minList (x:xs) k = minList xs (\v -> mink v x id)
```

**Problem 6)** For decList, maxList, and minList, explain why they are not in continuation passing style.

---

\(^1\)Well, it returns unit in the IO monad.
Conversion

Here are some of the conversion rules.

\[ C[f \ arg = e] \Rightarrow f \ arg \ k = C[e]_k \]

- **arg is simple**
  \[ C[a]_k \Rightarrow k \ a \]
  \[ C[f \ arg]_k \Rightarrow f \ arg \ k \]

- **\(e_1, e_2\) are simple**
  \[ C[e_1 + e_2]_k \Rightarrow k(e_1 + e_2) \]

- **\(e_2\) is simple**
  \[ C[e_1 + e_2]_k \Rightarrow C[e_1]_(\lambda v. f v k) \text{ where } v \text{ is fresh.} \]

**Problem 7)** The phrase ```where v is fresh``` appears a lot here. Why do we need to be concerned about that?

**Problem 8)** Suppose we want to convert these functions into CPS. There are helper functions \(f, g, \text{ and } h\). During the conversion process, we are also going to convert \(f\) and \(g\) to CPS, but leave \(h\) in direct style.

\[
\begin{align*}
0 & \text{ foo a b = f a + g b} \\
1 & \text{ bar x y = h x + y} \\
3 & \text{ baz c = 3 + c} \\
5 & \text{ quux d = h (g d)}
\end{align*}
\]

Which subexpressions in the code above are simple? Can you think of way to describe what being simple means in this context?
Convert To

**Problem 9)** Convert \texttt{map} to CPS. Assume \texttt{f} is written in direct style.

\begin{itemize}
  \item [0] \texttt{map f [] = []}
  \item [1] \texttt{map f (x:xs) = f x : map f xs}
\end{itemize}

**Problem 10)** Do it again, but this time assume \texttt{f} is written in CPS and takes one continuation.

\begin{itemize}
  \item [0] \texttt{map f [] = []}
  \item [1] \texttt{map f (x:xs) = f x : map f xs}
\end{itemize}

**Problem 11)** Convert the following code to CPS, preserving the order of operations that would be used if Haskell were an eager language. Note: you will need a \textit{nested continuation} to make this work.

\begin{itemize}
  \item [0] \texttt{min a b = if a < b then a else b}
  \item [1] \texttt{min4 a b c d = min (min a b) (min c d)}
\end{itemize}
Reordering Computations

On the off chance we have extra time, here’s something to try.

Suppose you have a calculator which has an accumulator and a list of instructions. Add \( i \) adds \( i \) to the accumulator, and Sub \( i \) subtracts \( i \) from the accumulator.

\[
\text{data } \text{Calc} = \text{Add Integer} \\
| \quad \quad \text{Sub Integer} \\
\text{deriving (Eq,Show)}
\]

The only problem is that our accumulator cannot ever be negative! Use continuations to fix this.

Here’s the original calculator:

\[
\text{calc } xx = \text{aux } \emptyset \; xx \\
\text{where } \text{aux } a \; [] = a \\
\quad \text{aux } a \; ((\text{Add } i):xs) = \text{aux } (a+i) \; xs \\
\quad \text{aux } a \; ((\text{Sub } i):xs) = \text{aux } (a-i) \; xs
\]

Hint: you will need two continuations to make this work.