Homework 6: Continuations

Jack Shi - A92122910
Wyatt Guidry - A12994977
Austin Coleman - A12888539

March 16, 2018

1 Why CPS?

1. **Answer:**

```
fun div(x, y, cc) {
  cc(x / y)
}

fun f(x, y, cc) {
  div(2, y, r => cc(3 * y * r))
}
div(3, 4, r => f(r, 5, top_cc))
```

2. β, η optimization
   For part **a** and **b**, first transform the functions into λ-calculus form.

$$div \equiv \lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y))$$
$$f \equiv \lambda x.\lambda y.\lambda cc.((div)\ 2\ y\ \lambda r.((cc)\ (3 * y * r)))$$
$$(div)\ 3\ 4\ \lambda r.((f)\ r\ 5\ top\_cc)$$
$$(\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((f)\ r\ 5\ top\_cc)$$

   (a) **Answer:** For the sake of clarity and readability, I did not rewrite multiplication as prefix function application.
   First simplify f:

$$\begin{aligned} f \equiv\ & \lambda x.\lambda y.\lambda cc.((div)\ 2\ y\ \lambda r.((cc)\ (3 * y * r))) \\ =\ & \lambda x.\lambda y.\lambda cc.((\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 2\ y\ \lambda r.((cc)\ (3 * y * r))) \\ =_\beta\ & \lambda x.\lambda y.\lambda cc.(\lambda r.((cc)\ (3 * y * r))\ (/\ 2\ y)) \\ =_\beta\ & \lambda x.\lambda y.\lambda cc.(\lambda r.((cc)\ (3 * y * r))\ (/\ 2\ y)) \\ =_\beta\ & \lambda x.\lambda y.\lambda cc.((cc)\ (3 * y * (/\ 2\ y))) \end{aligned}$$

   Subsitute `f` in to the original equation.

$$\begin{aligned} & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((f)\ r\ 5\ top\_cc) \\ =\ & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((\lambda x.\lambda y.\lambda cc.((cc)\ (3 * y * (/\ 2\ y))))\ r\ 5\ top\_cc) \\ =_\beta\ & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((top\_cc)\ (3 * 5 * (/\ 2\ 5))) \\ =_\beta\ & (\lambda r.((top\_cc)\ (3 * 5 * (/\ 2\ 5))))\ (/\ 3\ 4) \\ =_\beta\ & (top\_cc)\ (3 * 5 * (/\ 2\ 5)) \\ =\ & (top\_cc)\ (6) \end{aligned}$$

(b) **Answer:** For the sake of clarity and readability, I did not rewrite multiplication as prefix function application.

Simplify f using η reduction:

$$\begin{aligned} f \equiv\ & \lambda x.\lambda y.\lambda cc.((div)\ 2\ y\ \lambda r.((cc)\ (3*y*r))) \\ =\ & \lambda x.\lambda y.\lambda cc.((\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 2\ y\ \lambda r.((cc)\ (3*y*r))) \\ =_\eta\ & \lambda x.\lambda y.\lambda cc.((\lambda x.\lambda cc.(cc\ (/\ x\ y)))\ 2\ \lambda r.((cc)\ (3*y*r))) \\ =_\beta\ & \lambda x.\lambda y.\lambda cc.(\lambda r.((cc)\ (3*y*r))\ (/\ 2\ y)) \\ =_\beta\ & \lambda x.\lambda y.\lambda cc.((cc)\ (3*y*(/\ 2\ y))) \end{aligned}$$

By η reduction, `f` converged to the same result. The rest of the reduction follows the same as part **a.**.

$$\begin{aligned} & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((f)\ r\ 5\ top\_cc) \\ =\ & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((\lambda x.\lambda y.\lambda cc.((cc)\ (3*y*(/\ 2\ y))))\ r\ 5\ top\_cc) \\ =_\beta\ & (\lambda x.\lambda y.\lambda cc.(cc\ (/\ x\ y)))\ 3\ 4\ \lambda r.((top\_cc)\ (3*5*(/\ 2\ 5))) \\ =_\beta\ & (\lambda r.((top\_cc)\ (3*5*(/\ 2\ 5))))\ (/\ 3\ 4) \\ =_\beta\ & (top\_cc)\ (3*5*(/\ 2\ 5)) \\ =\ & (top\_cc)\ (6) \end{aligned}$$

3. Error throwing and catching

(a) **Answer:**

```
fun div(x, y, cc_ok, cc_fail) {
  if (y != 0) {
    cc_ok(x / y)
  } else {
    cc_fail(''Divide by zero'')
  }
}

fun f(x, y, cc) {
  div(2, y, r => cc(3 * y * r), r => 0)
}

div(3, 4, r => f(r, 5, top_cc_ok), top_cc_fail)
```

(b) **Answer:** In case of `Node.js`, when the call back begin to execute, the high level `try/catch` block is not responsible for handling the callback errors. The issue is similar here. If throw is used, then the high level `try/catch` might not catch if the function that throw an error is used as an callback for some function in the `try/catch` block.

2 Tail Recursion and Continuations

1. **Answer:** With compiler optimizations, since the recursive call is the last instruciton of the current function, there is no need to keep the previous stack frame. This allows tail recursive calls to all be executed in a single stack space which means that the space requirement is independent from n which governs the number of recursive calls.

2. **Answer:** This program require a stack for `fact`, a stack for tail recursive `f` call and n stacks for `g`. The stack for `fact` can be freed when the function is done. The stack for `f` is reused because of tail recursive optimization. The memory is freed right before `fact` stack is freed. The n `g` stacks are spawned when base case is hit and destroyed when the value is calculated for the base case.