Homework 5: Continuations

CSE 130: Programming Languages

Early deadline: September 3 23:59, Hard deadline: September 5, 23:59

Names & IDs:

1 [20pts] Why CPS?

Functional programming language compilers typically transform a source program into a λ-calculus-like language, which they then transform into continuation passing style (CPS). One of the beauties of CPS is that both control flow and data flow are represented using a single intermediate language with well-defined semantics. In this question we are going to investigate several benefits of CPS and the motivation for using CPS as an intermediate language.

We are going to a use a λ-calculus-like language, with JavaScript-like syntax, as our source language. We assume that primitives, such as addition and multiplication exist and are baked into the intermediate language.

1. [6pts] Recall that a CPS-transformed function takes an extra argument—the *continuation*. And, when invoking such a function the caller must provide a function which is to be invoked with the "`return`" value. Transform the following program into CPS. Assume that any sequence of primitive operations on variables (e.g., `c * d / e`) executes atomically and do not need to be CPS transformed. (An alternative, more standard approach would provide CPS transformed versions of the primitives; for simplicity we do no take this approach.) Additionaly, assume that the top level continuation is called `top_cc`.

```
fun div(x, y) {
  x / y
}
fun f(x, y) {
  3 * y * div(2, y)
}
f(div(3,4), 5))
```

Answer:

One interesting point you should notice about the transformation is that in addition to making control flow explicit, CPS also makes the order of evaluation explicit. In particular, notice that arguments to function are "trivial" expressions (or primitive operations on such expressions), and cannot be applications themselves.

2. [6pts] Once a program is in CPS, a compiler can perform various optimizations. For example, it is common, at this point, to do partial evaluation, i.e., evaluate certain expressions at compile time instead of run time,[1], tail-call elimination (replace calls with jumps), etc. Interestingly you can also perform β- and η reductions on the CPS intermediate language as a form of optimization. Recall that β and η are defined as:

   - $(\lambda x.e_1)\ e_2 =_\beta e_1[x := e_2]$
   - $\lambda x.e\ x =_\eta e$ if $x \notin FV(e)$

   For the questions below, assume that our compiler only handles single-argument functions, much like Haskell, and multiple-argument function are implemented by currying. Concretely, you should treat `function f(x, y, z) { e }` as $f \equiv \lambda x.\lambda y.\lambda z.e$.

   (a) [3pts] Apply a single β-reduction to optimize your CPS program, if possible.

   Answer:

   (b) [3pts] Apply a single η-reduction to optimize your CPS program, if possible.

   Answer:

[1] At this point you may be wondering what's the big deal, we do this for imperative languages—it's just constant folding! Unlike CPS, most representations (e.g., for C you would use triples: the operation and two arguments) do not have such beautiful semantics, nor can they be directly executed.

3. [8pts] Since CPS makes control flow explicit, supporting complex control flow constructs, such as function returns, exceptions, and cooperative concurrency is mostly straight forward. To see this, let's modify our source program to add conditionals and exceptions:

```
fun div(x, y) {
  if (y != 0) {
    x / y
  } else {
    throw "Divide by zero";
  }
}
fun f(x, y) {
  try {
    3 * y * div(2, y)
  } catch (e) {
    0
  }
}
f(div(3,4), 5)
```

(a) [6pts] CPS transform this program by passing the exceptional continuation (when necessary) alongside the "normal" (or success) continuation (e.g., as done in part 1, above) for the expression that throws the exception. Interestingly, the desugaring of the `try {...} catch {...}` block just desugars into calling the function (that may raise an exception) with the two different continuations. For this question you can assume that our intermediate language has `if` expressions built-in (where the branch condition can be any "trivial" expression). Moreover, assume that the top-level success continuation is called `top_cc_ok`, while the failure continuation is called `top_cc_fail`.

Answer:

(b) [2pts] The desugaring of the `if` condition above was straight forward since we assumed that the CPS intermediate language had `if` expressions. Suppose that our CPS intermediate language also had exception constructs, would it be safe to transform the `throw` expression into a similarly straight forward way, by raising an exception in the CPS-transformed `div` function? If so, why? If not, why not? *Hint: consider what would happen if a callback in Node.js threw an exception.*

Answer:

2 [Optional]Tail Recursion and Continuations

1. [5pts] Explain why a tail recursive function, such as

```
function fact(n){
    function f(n,a){
        if (n==0) return a
        else return f(n-1, a*n)
    }
    return f(n,1)
};
```

can be compiled so that the amount of space required to compute `fact(n)` is independent of `n`.

Answer:

2. [7pts] The function `f` used in the following definition of factorial is "formally" tail recursive: the only recursive call to `f` is a call that need not return.

```
function fact(n){
    function f(n,g){
        if (n==0) return g(1);
        else return f(n-1, function(x){return g(x)*n})
    };
    return f(n, function(x){return x})
};
```

How much space is required to compute `fact(n)`, measured as a function of argument `n`? Explain how this space is allocated during recursive calls to `f` and when the space may be freed.

Answer: