Classical Logic: Lies, Trickery, and Continuations

Michael Coblenz (mcoblenz@cs.cmu.edu)

September 29, 2015

1 Continuations, continued

In the previous lecture, we saw how classical logic permits trickery. If we allow ourselves to use that trickery, we can write some interesting programs.

In order to understand this better, I'd like to delve more deeply into continuations.

Let's implement some functions in continuation-passing style. We'll start with a regular function that computes the length of a two-dimensional vector:

```
fun length x y =
    Math.sqrt (x*x + y*y)
```

Now, let's assume that this length calculation is one step in a whole sequence of computations. After we compute the length of the vector, there's more work to do. We'll assume there's a function that takes the length as input and then does the rest of the work:

```
fun lengthCont x y k =
    k (Math.sqrt (x*x + y*y))
```

Now, let's restructure this to make order of evaluation explicit.

```
fun lengthCont2 x y k =
    (fn x2 =>
        (fn y2 =>
           (fn x2py2 =>
               (fn len => k len)
           (Math.sqrt x2py2)
           ) (Real.+(x2, y2))
        ) (Real.*(y, y))
    ) (Real.*(x, x))
```

How does this make order of evaluation explicit? This is a call-by-value language, so before calling a function, we must evaluate its argument down to a value. All the computation is done one step at a time in the arguments to functions.

An exercise: implement a continuation-passing style factorial function.

```
fun factCont n k =
    (fn eqZero => if eqZero then (k 1) else
        (fn nMinusOne => factCont nMinusOne
            (fn factNMinusOne =>
                 (fn result => k result
                 ) (factNMinusOne * n)
            )
        ) (n - 1)
   ) (n = 0)
```

2 On Pierce's Law

We learned before that Pierce's Law is:

$$((A \supset B) \supset A) \supset A$$

and that its proof term is

$$\lambda f.Ck.\textbf{throw}\ k\ (f\ (\lambda u.\textbf{throw}\ k\ u))$$

Suppose we have this proof term. Let's give it a name, *call*/*cc*, which stands for "call with current continuation." What is f and what can we do with this? And what is the B in the type?

Let's look at what f gets applied to: $(\lambda u.\textbf{throw}\ k\ u)$. Question: what are all the types?

$$u : A$$

$$(\lambda u.\textbf{throw}\ k\ u) : A \supset B$$

What is B? It doesn't matter because this function takes u and **never returns**. Can you think of an example of a C function that never returns? Hint: there is at least one in the standard library, and if you've programmed in C, you've probably used it before.

$$f : (A \supset B) \supset A$$

Now, what are k and Ck? Think of C as a binding operator, like λ. k is the name of the bound variable. k means "the current state." k allows us to time-travel back to where we were at the time it was bound! When a value is thrown

to k, time rewinds back to Ck and we proceed as if nothing else had happened. For example, suppose we had committed to returning **inr** of something. We could change our minds and return **inl** of something instead! This is exactly what the law of the excluded middle does.

$$LEM = Ck : A \vee \neg A.\mathbf{throw}^{\#} k(\mathbf{inr}^{A}(\lambda v : A.\mathbf{throw}^{\perp} k(\mathbf{inl}^{\neg A} v)))$$

What does it mean for a function (as in **throw** above) to return type $\perp$? If the function returned, it would prove $\perp$, which is impossible, so this must mean that the function never returns.

3 Reduction

To reduce around a continuation, we need to push operations inside it. Take this example:

$$(Ck.\mathbf{throw}\ k\ M)\ N \Rightarrow_R Ck.\mathbf{throw}\ k\ (M\ N)$$

Why is this right? We started with an expression that captured the state and threw M to that state; then it applied that to N. That's the same as first applying M to N and then doing that in the current state.

Let's consider another example:

$$\mathbf{snd}\ (Ck.\mathbf{throw}\ k\ M) \Rightarrow_R Ck.\mathbf{throw}\ k\ (\mathbf{snd}\ M)$$

This is the same: push the snd inside the thing we're going to throw. What's the general pattern? If we want to do an operation on a proof by contradiction, we instead do the operation on the thing that was going to get thrown. From the notes: The syntax $[k'\ (\mathbf{snd}\ \Box)/k]E$ stands for a new kind of substitution, structural substitution into the body of E. Roughly speaking, in the body of E, every instance of $\mathbf{throw}\ k\ M$ is replaced with $\mathbf{throw}\ k'\ (\mathbf{snd}\ \Box)$. The box $\Box$ stands for the "hole" where the old argument to throw, M, gets placed. The label k' corresponds to the proposition proved by the elimination rule.

There's one more kind of reduction to consider. What if we apply an operator to a throw? That has no point because the throw never returns. So, for example:

$$(\mathbf{throw}^{A \supset B}\ k\ M)\ N \Rightarrow_R (\mathbf{throw}^{B}\ k\ M)$$