15-399 Supplementary Notes: The Computational Content of Classical Logic

Robert Harper

February 18, 2003

1 First-Class Continuations

Standard ML of New Jersey extends the Standard ML language with *first-class continuations*. These form an abstract type with the following signature:

```
signature CONT =
sig
  type 'a cont
  val callcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
end
```

The type `'a cont` is the type of *continuations* accepting values of type `'a`. Continuations are created using `callcc`, and a value is passed to a continuation using `throw`.

In general a continuation represents an *evaluation point* in an expression. An evaluation point, indicated by a "hole" ($\Box$) in an expression, represents a checkpointed computation that may be resumed by providing a value for that spot in the expression. Evaluation continues from that point onwards according to the normal rules of evaluation. An expression with a hole in it marking an evaluation point is called a *continuation*. The hole in a continuation may be filled using `throw`, which provides a value for the hole and resumes execution from that point. Throwing a value to a continuation *abandons* the current evaluation, effecting a "goto" the continuation to which the value is passed. Continuations are created using `callcc`, which seizes the current continuation (the evaluation point at which the application of `callcc` occurs), and passes it to a given function.

The use of continuations in a program is best explained by example. Consider the following simple function to multiply together the integers in a list:

```
fun mult l =
  let
    fun loop nil => 1
      | loop (h::t) => h * loop t
  in
    loop l
  end
```

This function defines an inner loop that performs the multiplication.

Suppose that we wish to “short circuit” `mult` in the case that zero occurs in the list. How do we exit the function early with the value zero, avoiding the need to perform any further multiplications? One method is to grab the continuation at the point where the function is just about to return, so that we can throw zero to it when we wish to exit early. This is achieved using `callcc` and `throw` as follows:

```
fun mult l =
  callcc
    (fn ret =>
      let
        fun loop nil => 1
          | loop (0::_) => throw ret 0
          | loop (h::t) => h * loop t
      in
        loop l
      end)
```

The continuation bound to `ret` corresponds to the point right after the equal sign, which is the point at which the value of the function is returned to the caller. Should zero occur at some point in the list, control jumps immediately to this point with the value `0`, effecting an early return of the function.

One thing to note about the function `mult` is that it passes a value to the return point of the function in two distinct ways: explicitly, by throwing 0 to it when 0 arises in the list, and implicitly, by simply returning the value of `loop l`. It is entirely equivalent to replace the second-to-last line of `mult` with this one:

```
throw ret (loop l)
```

In fact there is no loss of generality in *insisting* that the body of the `callcc` throw to some continuation, rather than returning normally. This may may be achieved by replacing `callcc` with `callcc'` whose type is `('a cont -> void) -> 'a`. The type `void` is defined by the following signature:

```
signature VOID =
  sig
    type void
    val abort : void -> 'a
  end
```

The type `void` has no elements, so a function of type `'a cont -> void` cannot return — it must `throw` a value to a continuation. Indeed, if we modify the short-circuit version of `mult` as just indicated, then `callcc` may be replaced by `callcc'`, and the program will still type check.

2 Continuations and Negation

Recall that $\neg P$ may be thought of as $P \supset \perp$, either informally as a guide to the primitive rules of negation, or formally, by defining negation in terms of implication and falsehood. If we think of $\neg P$ in this way, then we see that a proof of a negated proposition is a function that never returns — for if it were to return, it would have to return a proof of $\perp$, which is impossible.

Continuations may also be thought of as functions that never return. When we throw a value to a continuation, it jumps to that continuation and never returns to the point at which the throw occurred. This suggests that the type `'a cont` may be thought of as `'a -> void`, and that `throw` k v may be thought of as $\mathit{abort}(k(v))$. Since `void` corresponds to $\perp$ under the Curry-Howard isomorphism, this suggests that P `cont` corresponds to $\neg P$ — continuations are the computational content of refutations (proofs of negations)!

Under this interpretation the type of `callcc'` is

```
(('a -> void) -> void) -> 'a.
```

Look familiar? Written in logical notation, this is

$$((P \supset \perp) \supset \perp) \supset P.$$

In other words `callcc'` is the computational content of the law of double negation elimination!

3 The Law of the Excluded Middle

In the presence of double negation elimination we may prove the law of the excluded middle. This means that we can find a proof term of type $P \vee \neg P$ using `callcc'`. We will obtain it in two steps. First, recall that we can prove constructively $\neg\neg(P \vee \neg P)$. Here is a proof term of that type:[1]

$$M = \lambda k{:}\neg(P \vee \neg P).k(\mathbf{inr}(\lambda u{:}P.k(\mathbf{inl} u))).$$

Let M_{LEM} be the term `callcc'`(M) of type $P \vee \neg P$.

Let us analyze the computational behavior of this term. Since M_{LEM} is a proof of a disjunction, it may be used as the subject of a case analysis of the form

$$\mathbf{case}\, M_{LEM} \,\mathbf{of}\ \mathbf{inl}(u) \Rightarrow A | \mathbf{inr}(v) \Rightarrow B.$$

[1]Here we identify $\neg P$ with $P \supset \perp$, rather than treat negation as a primitive notion.

The term M_{LEM} seizes the current continuation, and passes it as argument to the term M given above. The current continuation is the function

$$K = \lambda x.\mathbf{case}\, x \ \mathbf{of}\ \mathbf{inl}(u) \Rightarrow A|\mathbf{inr}(v) \Rightarrow B$$

that, when applied, performs a case analysis on its argument. Applying M to K results in evaluation of the expression

$$K(\mathbf{inr}(\lambda u{:}P.K(\mathbf{inl}u))),$$

which in turn leads to the evaluation of

$$[\lambda u{:}P.K(\mathbf{inl}u)/v]B.$$

Now what can B do with v? Since its type is $\neg P = P \supset\perp$, the only thing that B can do with it is to apply it to a proof Q of P. Doing so leads to evaluation of

$$K(\mathbf{inl}Q)$$

which in turns leads to

$$[Q/u]A.$$

What is going on here? The idea is this. First off, M_{LEM} calls its continuation with a proof of $\neg P$. That is, when asked whether P or $\neg P$ is true (one or the other must be, since it is a proof of $P \vee \neg P$), it answers "$\neg P$", providing the proof $\mathbf{inr}(\lambda u{:}P.K(\mathbf{inl}u))$. If B ever makes use of this proof, it can only do so by providing a proof of P, contradicting the purported proof of $\neg P$ provided to M_{LEM}'s continuation! But then the proof of $\neg P$ rescues itself by *backtracking*, abandoning the $\neg P$ case, and instead invoking the P case with the given proof of P. In other words, first it "lies" by asserting that $\neg P$ is true. If it "gets caught", then it backtracks, and says "oh, I meant P all along, sorry about that." From there onwards the proof proceeds as normal.

What makes this possible is first-class continuations. Crucially, the continuation K above is used *twice*, once to pass the "proof" of $\neg P$, then again (if necessary) with the given proof of P. Without continuations this cannot be programmed.[2]

[2]Proof: there is no normal proof of the law of the excluded middle.