15-399 Supplementary Notes: Classical Logic, Computationally

Robert Harper

April 13, 2005

Classical Logic

The meaning of a proposition in classical logic is ordinarily given *denotationally* by saying that a proposition stands for either $\top$ or $\bot$. The logical connectives are defined by truth tables displaying the graph of their associated functions on truth values. The denotational semantics of classical logic emphasizes the "absolute" truth or falsehood of a proposition, independently of how we, as humans, may come to learn it.

In contrast, the meaning of a proposition in constructive logic is given *operationally* by defining what counts as a proof of it. The logical connectives are defined by rules for proving propositions formed with each connective. The operational semantics of constructive logic emphasizes the process of learning and communicating the truth of a proposition in a proof.

The close connection between constructive logic and computer programming stems from its operational semantics. As we have seen, a proposition can be read as a specification, or problem statement, and a proof as a program, or solution to that problem. Classical logic, with its emphasis on absolute truth, ignores the operational aspects of proof, and consequently neglects the connection with programming.

Despite the difference in semantics, the social processes of classical and constructive mathematics are largely the same. In particular, the conduct of classical mathematics centers on the communication of knowledge through proof, just as in constructive mathematics. This suggests that there may be an alternative semantics for classical logic that exposes its operational content, thereby establishing a connection to computer programming. The purpose of this note is to explore this connection.

Proof by Contradiction

Viewed operationally, the distinction between classical and constructive logic comes down to the acceptance (classical logic) or denial (constructive logic) of

the principle of *proof by contradiction*. In classical logic we may prove a proposition P by *assuming* $\neg P$ and deriving a contradiction.[1] From a constructive viewpoint we are only entitled to conclude $\neg\neg P$, since we have refuted the assumption of $\neg P$, but with classical logic this is the same as affirming P itself.

There are many ways to formalize this principle of reasoning. The one we shall adopt here is to generalize the notion of a hypothetical judgment to permit two forms of assumptions, the familiar *truth* assumptions, plus a new form, the *falsehood* assumptions. These judgments have the form

$$\underbrace{\ldots, x_i : P_i\ \mathit{true}, \ldots}_{\Gamma}; \underbrace{\ldots, u_j : Q_j\ \mathit{false}, \ldots}_{\Delta} \vdash R\ \mathit{true}$$

asserting the truth of R under the assumptions that the P_i's are true and that the Q_j's are false.

Using this generalized form of hypothetical judgment we may formulate the principle of *proof by contradiction* as follows:

$$\frac{\Gamma; \Delta, u : P\ \mathit{false} \vdash P\ \mathit{true}}{\Gamma; \Delta \vdash P\ \mathit{true}}$$

Note well that in the premise of the rule P is assumed to be *false* while deriving its own truth! This corresponds to the informal practice of assuming $\neg P\ \mathit{true}$ while deriving P from it. The only way to use an assumption of falsehood is to contradict it, concluding whatever we like:

$$\frac{\Gamma; \Delta', u : P\ \mathit{false} \vdash P\ \mathit{true}}{\Gamma; \Delta', u : P\ \mathit{false} \vdash Q\ \mathit{true}.}$$

Just as with constructive logic, we may assign proof terms to derivations. Here are the two rules for classical logic written using proof terms:

$$\frac{\Gamma; \Delta, u : P\ \mathit{false} \vdash M : P}{\Gamma; \Delta \vdash \mathbf{letcc}\, u\, \mathbf{in}\, M : P}$$

$$\frac{\Gamma; \Delta', u : P\ \mathit{false} \vdash M : P}{\Gamma; \Delta', u : P\ \mathit{false} \vdash \mathbf{throw}\, M\, \mathbf{to}\, u : Q.}$$

It is interesting to derive a proof term for the law of the excluded middle, which is provable in classical logic. The required term is

$$\mathbf{letcc}\, u\, \mathbf{in}\, \mathbf{inr}(\lambda x{:}P.\ \mathbf{throw}\, \mathbf{inl}(x)\, \mathbf{to}\, u)$$

whose type is $P \vee \neg P$, as required. It is an easy exercise to check that this term has the required type, but what does it mean? To answer this we must tease out the computational content of classical proofs.

[1] In fact it is enough to derive P from the assumption of $\neg P$, for if we do so, then we have a contradiction, and if we have a contradiction from the assumption of $\neg P$, then we can conclude, in particular, P, since anything follows from a contradiction.

Evaluation of Classical Proofs

In constructive logic the inversion principle — stating that the elimination rules are inverse to the corresponding introduction rules — gives rise to the concept of proof reduction, or simplification, in which we may "cancel" any use of an elimination form whose principal argument is a corresponding introduction form. Proof terms may be viewed as functional programs that are evaluated by performing reduction in a particular order, such as the call-by-value ordering used by ML.

This connection extends also to classical logic, resulting in a functional programming language with so-called *control operators*. To see how these arise, let us consider the sense in which the rule for eliminating (using) false assumptions (the **throw** construct) is inverse to the rule for introducing them (the **letcc** construct). The typing rule for **letcc** states that $\mathbf{letcc}\, u\, \mathbf{in}\, M$ is a proof of A provided that M is a proof of A *under the additional assumption, u, that A is false*. Now it is entirely possible that M makes no use of u at all, and is a direct proof of A all by itself. In that case the use of **letcc** is spurious, and we can simplify using the rule

$$\mathbf{letcc}\, u\, \mathbf{in}\, M \rightsquigarrow M \qquad (u \notin \mathrm{FV}(M)).$$

The important case is that M makes use of u. Since u is a false assumption, the only possible way to use it is with the **throw** construct. This means that $\mathbf{letcc}\, u\, \mathbf{in}\, M$ has the form

$$\mathbf{letcc}\, u\, \mathbf{in} \ldots \mathbf{throw}\, N\, \mathbf{to}\, u \ldots$$

for some proof N of A. For simplicity, let us assume that $u \notin \mathrm{FV}(N)$, but in general we must take account of this possibility as well. This means that buried within M we actually have in hand a *direct* proof, N, of A — one that does not make use of the assumption that u is false. This suggests that we may cancel the **letcc** using the following rule:

$$\mathbf{letcc}\, u\, \mathbf{in} \ldots \mathbf{throw}\, N\, \mathbf{to}\, u \ldots \rightsquigarrow N.$$

Note that both sides of this simplification are proofs of the same proposition A! The interesting thing about this rule is that it is a *non-local* simplification in the sense that it is not based on a cancellation of an *adjacent* pair of constructs, but rather on a *nesting* of a *throw* inside the scope of the variable to which it throws. This rule has the flavor of a "goto" construct, in which u plays the role of a label, and the **throw** construct constitutes a "goto with an argument" to the label u. In particular, the ellided parts of M (represented here by the "...") are abandoned by the simplification step.

Unfortunately, the story is not quite that simple. Whether or not the above rule applies depends crucially on what is ellided; we may not apply this rule for any choice of "...", but only in certain circumstances. To see what can go wrong, suppose that A has the form $A_1 \supset A_2$, and consider the following proof of A:

$$\mathbf{letcc}\, u\, \mathbf{in}\, \lambda x{:}A_1.\mathbf{throw}\, N\, \mathbf{to}\, u.$$

The typing rule ensures that N is a proof of A, but what if x occurs free in N? If so, it makes no sense to reduce this term to N, because we have no binding for x! Rather, we must "wait" for the argument x to come along before we can execute the **throw**. In other words we must impose a *reduction strategy* on proof terms to ensure that the proof reductions happen "in the right order" to avoid such anomalies.

To make these ideas precise, we must define a notion of reduction that is *deterministic* (simplifies in a specific order) and *contextual* (keeps track of the context in which the simplification occurs). This is achieved using *evaluation contexts*, which are defined as follows:

$$\begin{array}{rcl} \mathcal{E} & ::= & \circ \mid \langle \mathcal{E}, M\rangle \mid \langle V, \mathcal{E}\rangle \mid \mathbf{fst}(\mathcal{E}) \mid \mathbf{snd}(\mathcal{E}) \mid \\ & & \mathcal{E}\, M \mid V\, \mathcal{E} \mid \mathbf{abort}(\mathcal{E}) \mid \mathbf{inl}(\mathcal{E}) \mid \mathbf{inr}(\mathcal{E}) \mid \\ & & \mathbf{case}\mathcal{E}\ \mathbf{of}\ \mathbf{inl}(u{:}P) \Rightarrow N_1 \mid \mathbf{inr}(v{:}Q) \Rightarrow N_2 \end{array}$$

Here we use the letter V to range over *values*, defined as follows:

$$V ::= \quad x \mid \langle V_1, V_2\rangle \mid \lambda u{:}P.M \mid \mathbf{inl}(V) \mid \mathbf{inr}(V)$$

Values are terms that need no further simplification, even though they may contain reducible sub-expressions within them (in particular, inside the body of a λ-abstraction).

Evaluation contexts are used to represent the surrounding context of an instruction step. An evaluation context is essentially an expression with a "hole" in it, written "$\circ$". The hole represents the spot at which execution is occurring. If $\mathcal{E}$ is an evaluation context, we write $\mathcal{E}\{M\}$ for the result of replacing the hole in $\mathcal{E}$ with M; we say that $\mathcal{E}$ is "filled" by M. An evaluation context, $\mathcal{E}$, may be treated as a value, written $\hat{\mathcal{E}}$, that may be substituted for a variable during evaluation.

We may then define evaluation of terms by the following transitions:

$$\begin{array}{rcl} \mathcal{E}\{\mathbf{fst}(\langle V_1, V_2\rangle)\} & \mapsto & \mathcal{E}\{V_1\} \\ \mathcal{E}\{\mathbf{snd}(\langle V_1, V_2\rangle)\} & \mapsto & \mathcal{E}\{V_2\} \\ \mathcal{E}\{(\lambda u{:}P.M)\, V\} & \mapsto & \mathcal{E}\{[V/u]M\} \\ \mathcal{E}\{\mathbf{letcc}\, u\, \mathbf{in}\, M\} & \mapsto & \mathcal{E}\{[\hat{\mathcal{E}}/u]M\} \\ \mathcal{E}\{\mathbf{throw}\, V\, \mathbf{to}\, \hat{\mathcal{E}'}\} & \mapsto & \mathcal{E}'\{V\} \end{array}$$

Observe that **letcc** captures the evaluation context $\mathcal{E}$ by substituting it for u in the body, and that **throw** abandons $\mathcal{E}$ in favor of the context $\mathcal{E}'$ passed to it. It is by this means that the "goto-like" behavior of the **throw** construct is obtained.

Translation to Constructive Logic

An indirect method for explaining the computational content of classical proofs is to translate classical proofs into constructive proofs. But how is this possible? The main idea was hinted at in the introduction — from a constructive

viewpoint, a proof by contradiction of P in classical logic is just a constructive proof of $\neg\neg P$.

This suggests that there may be a systematic way to "constructivize" classical logic, by revising the theorem into a form that is weaker, from a constructive viewpoint, but is classically equivalent. There are many ways to do this, all of which are loosely known as the *double-negation translation*, or the *Gödel-Gentzen translation*.

$$\begin{array}{rcl} \top^\circ & = & \top \\ (P \wedge Q)^\circ & = & P^\circ \wedge Q^\circ \\ (P \supset Q)^\circ & = & P^\circ \supset \neg\neg Q^\circ \\ (P \vee Q)^\circ & = & P^\circ \vee Q^\circ \\ \bot^\circ & = & \bot \end{array}$$

The translation is extended to contexts by defining $\Gamma^\circ(u) = \Gamma(u)^\circ$ for each assumption u in Γ. Similarly, we define the context $\neg\Gamma$ such that $(\neg\Gamma)(u) = \neg\Gamma(u)$ for each assumption u in Γ.

Theorem 0.1 *If $\Gamma; \Delta \vdash P$ true, then $\Gamma^\circ, \neg\Delta^\circ \vdash \neg\neg P^\circ$ true.*

The proof of this theorem is most easily given using proof terms.

Theorem 0.2 *If $\Gamma; \Delta \vdash M : P$, then $\Gamma^\circ, \neg\Delta^\circ \vdash M^* : \neg\neg P^\circ$ for some term M^*.*

Proof: The proof is by induction on the structure of the proof term M. We give some representative cases.

- Suppose that M is a variable u such that $\Gamma(u) = P$. Let $u^* = \lambda k{:}\neg P^\circ.k\,u$, and check that $\Gamma^\circ, \neg\Delta^\circ \vdash u^* : \neg\neg P^\circ$.

- Suppose that $P = Q \vee R$ and $M = \mathbf{inl}(N)$ with $\Gamma; \Delta \vdash N : Q$. By induction $\Gamma^\circ, \neg\Delta^\circ \vdash N^* : \neg\neg Q^\circ$. Let $M^* = \lambda k{:}\neg(Q^\circ \vee R^\circ).N^*(\lambda u{:}Q^\circ.k(\mathbf{inl}(u))$.

- Suppse that $M = \mathbf{case} N\ \mathbf{of}\ \mathbf{inl}(x{:}Q) \Rightarrow N_1 \mid \mathbf{inr}(y{:}R) \Rightarrow N_2$. By induction we have

  1. $\Gamma^\circ, \neg\Delta^\circ \vdash N^* : \neg\neg(Q^\circ \vee R^\circ)$;
  2. $\Gamma^\circ, \neg\Delta^\circ, x{:}Q^\circ \vdash {N_1}^* : \neg\neg P^\circ$;
  3. $\Gamma^\circ, \neg\Delta^\circ, y{:}R^\circ \vdash {N_2}^* : \neg\neg P^\circ$.

  Let M^* be the term

  $$\lambda k{:}\neg P^\circ.N^*\,(\lambda u{:}Q^\circ \vee R^\circ.\mathbf{case} u\ \mathbf{of}\ \mathbf{inl}(x{:}Q^\circ) \Rightarrow {N_1}^*\,k \mid \mathbf{inr}(y{:}R^\circ) \Rightarrow {N_2}^*\,k).$$

- Suppose that $M = \mathbf{throw}\,N\,\mathbf{to}\,u$, where $\Gamma; \Delta \vdash N : Q$ and $\Delta(u) = Q$. By induction we have $\Gamma^\circ, \neg\Delta^\circ \vdash N^* : \neg\neg Q^\circ$. Let $M^* = \lambda k{:}\neg P^\circ.N^*\,(\lambda x{:}Q^\circ.u\,x)$. Note that the parameter k of the λ-abstraction is ignored in its body!

- Suppose that $M = \mathbf{letcc}\, u\, \mathbf{in}\, N$, where $\Gamma; \Delta, u{::}P \vdash N : P$. By induction we have $\Gamma^\circ, \neg\Delta^\circ, u{:}\neg P^\circ \vdash N^* : \neg\neg P^\circ$. Let $M^* = \lambda k{:}\neg P^\circ.[k/u]N^*\, k$. Note that k is replicated by substitution, and also used as the argument to N^*!

□

This translation amounts to a "compiler" that turns a classical proof into a constructive one, thereby obtaining a proof term that expresses the computational content of the classical proof. This translation is, in fact, one used in "real" compilers (for example, the SML/NJ compiler); it is called the *cps translation*, where "cps" stands for "continuation-passing style". The main ideas of the cps translation is that the translation of a program fragment takes as an extra argument the "return address" to which to return its result. The return address for an expression of type P is a function, called a *continuation*, of type $\neg P = P \supset \bot$. Such a function never returns when called; instead it passes control to some other continuation, effectively "jumping" to that return address. The translation threads a continuation argument through the program, and arranges that any value which would ordinarily be "returned" is instead passed to the current continuation.

The cps translation has these three crucial properties:

1. It makes explicit the order of evaluation of sub-expressions of an expression. This corresponds to writing down a linear sequence of instructions to be executed to evaluate a program.

2. It binds every intermediate result to a variable before using it. This corresponds to putting an intermediate result into a machine register.

3. It makes the current continuation available as a value of function type so that it may be manipulated as data.

The translation reveals the computational behavior of **letcc** and **throw**:

- The **throw** construct *ignores* the current continuation, and instead passes a value to the continuation given to it as argument. This construct effectively "jumps" to a specified point in the program, and never returns to the point at which the **throw** occurs.

- The **letcc** construct *seizes* the current continuation by binding it to a variable for later use as argument to a **throw** expression. The current continuation is also used for the "normal" return of the body of the `letcc` in the case that it does not perform such a **throw**.

It is worthwhile to stare at the cps translation until this informal description becomes clear to you.

Notice that the context Δ of classical logic consists entirely of continuations; they are translated to functions of type $\neg P$, where P is their type in Δ. In other words, what classical logic gives you above and beyond constructive

logic is the ability to perform *non-local control transfers* to some continuation bound in Δ. In yet other terms, we may say that proofs in classical logic are functional programs (proofs in constructive logic) equipped with an additional "goto" construct!

Let M be the proof of the law of the excluded middle in classical logic, given above. What is its cps translation into constructive logic? After some simplification, it is the term[2]

$$\lambda k{:}\neg(P^\circ \vee \neg P^\circ).k\,(\mathbf{inr}(\lambda v{:}P^\circ.k\,(\mathbf{inl}(v)))).$$

How does this proof term execute? Since it is a proof of $P^\circ \vee \neg P^\circ$, its continuation is a function expecting a proof of $P^\circ \vee \neg P^\circ$. The proof passes to this continuation a proof of $\neg P$, marked as being injected into the right-hand side of the disjunction. This proof is a function that, when passed a proof, v, of P°, *abandons the current continuation* and instead passes v to the *original* continuation, k, marked as being injected into the left-hand side of the disjunction!

How does this work? To understand it, we must consider what the continuation k does with its argument, a proof of $P^\circ \vee \neg P^\circ$. If it makes use of it at all, it must perform a case analysis on it, and then branch according to whether the argument is injected into the left- or right-hand disjunct. *The current continuation is the point at which the proof is case analyzed.* The first time through the case analysis proceeds to the **inr** branch, obtaining in the process a proof of $\neg P^\circ$. If it ever uses this proof, it does so by passing to it a proof of P°. But then the proof *backtracks* to the point of the case analysis, this time with a left injection of that very proof! The case analysis now proceeds to the **inl** branch, without further difficulty. In effect, the proof of the law of the excluded middle *lies* by asserting $\neg P^\circ$, but then *escapes the consequences* of getting caught in a lie by backtracking to the point at which the lie was told, this time telling the truth!

[2] I am making implicit use of the fact that $\neg\neg\bot$ is equivalent to $\bot$ in order to avoid clutter, so that $(\neg P)^\circ$ may be taken to be $\neg P^\circ$.