Lectures 9 and 10: Classical Logic

15-317: Constructive Logic
Dan Licata

September 23-25, 2008

In these two lectures, we will discuss *classical logic*—which is what people were talking about when they taught you about (unqualified) "logic" in other classes—and its relationship to constructive logic—which is what we've covered so far this semester. We use *intuitionistic logic* as a synonym for "constructive logic"; this is helpful so we can use the abbreviations IL (intuitionistic) and CL (classical). We will answer three questions:

- What is classical logic?
- What is the relationship with intuitionistic logic?
- What is the computational meaning of classical proofs?

1 What is classical logic?

Classical logic differs from intuitionistic logic in that classical logic admits *the law of the excluded middle* (LEM), which says that every proposition is either true or false. The simplest way to describe classical logic is to take the natural deduction rules we have seen so far and add LEM:

$$\frac{}{(A \vee \neg A)\ \mathsf{true}}\ LEM$$

Equivalently, we could instead add *double-negation elimination*, which says that $\neg\neg A$ implies A:

$$\frac{\neg\neg A\ \mathsf{true}}{A\ \mathsf{true}}\ DNE$$

Note that double-negation introduction ($A \supset \neg\neg A$) is intuitionistically true.

It's easy to see that these are equivalent, by taking IL + LEM and deriving DNE and vice versa. For example, to derive DNE, assume $\neg\neg A$. Next, we use LEM on A to get $A \vee \neg A$, and therefore have two cases in which we have to show A. In the first, it's true by assumption. In the second, we have $\neg A$ and $\neg\neg A$, a contradiction, so we get A by $\bot$-elimination.

In the other direction, here is an annotated Tutch-style proof using DNE:

```
[u : ~(A | ~A)
 [v : A
  inl v : A | ~A
  u (inl v) : F]
 fn v => u (inl v) : ~A
 inr (fn v => u (inl v)) : A | ~A
 u (inr (fn v => u (inl v))) F]
fn u => u (inr (fn v => u (inl v))) : ~~(A | ~A)
DNE (fn u => u (inr (fn v => u (inl v)))) : (A | ~A)
```

A couple of things to note:

- Everything before the last line is a perfectly good constructive proof of `~~(A | ~A)`.
- The proof term `DNE (fn u => u (inr (fn v => u (inl v))))`. Doesn't tell you which of `A` or `~A` is true. Instead, it assumes `u : ~(A | ~A)` and proves a contradiction. It does this by calling `u` with a proof of `~A`. How does this proof work? When given an `A`, it calls `u` again, this time with that very proof that it was given! To anthropomorphize a little: the first time we call `u`, we bluff that the answer is `~A`. If `u` ever calls our bluff, it must do so by giving us an `A` and requesting a proof of `F`. In that case, we say "dang, you caught me! forget about what I said before—it was `A` after all!". We will talk more about this "time travel" interpretation of classical proofs below.

1.1 A Better Proof Theory

Hopefully you had a bad feeling about the rules LEM and DNE above: they violate some of our principles for what inference rules should look like. In particular, they mention more than one connective, and they are neither intro nor elim rules. Violating these principles can invalidate local soundness and completeness (and their global versions, cut elimination and identity). In this section, we give a cleaner presentation of classical logic by accounting for DNE at the level of judgments, rather than propositions.

1.1.1 New judgments

To do so, we need two new judgments:

- $\#$ (contradiction)
- $A\,\mathsf{false}$

The judgment $\#$is a judgmental analogue of $\bot$, the false proposition, whereas $A\,\mathsf{false}$ is a judgmental analogue of $A \supset \bot$. The rules for these judgements are as follows:

$$\frac{\#}{J}\,\#E \qquad \frac{\begin{array}{c}\overline{A\,\mathsf{true}}^{\,u}\\ \vdots\\ \#\end{array}}{A\,\mathsf{false}}\,fI^u \qquad \frac{A\,\mathsf{false} \quad A\,\mathsf{true}}{\#}\,fE$$

The first says that from a contradiction, you can conclude any judgment J. The second says that A is false if assuming it's true gives a contradiction. The third says that A being both true and false is contradictory.

A small technical matter: because we now have multiple judgments, we need to change the rules we had before that conclude an arbitrary proposition $C\,\mathsf{true}$ ($\bot$-elimination, $\vee$-elimination) to instead conclude an arbitrary *judgment* J. I.e.

$$\frac{\bot\,\mathsf{true}}{J}\,\bot E \qquad \frac{A \vee B\,\mathsf{true} \quad \begin{array}{c}\overline{A\,\mathsf{true}}^{\,u}\\ \vdots\\ J\end{array} \quad \begin{array}{c}\overline{A\,\mathsf{true}}^{\,u}\\ \vdots\\ J\end{array}}{J}\,\vee E^{u,v}$$

1.1.2 Negation

Using these judgments, we can give a primitive account of negation, rather than treating $\neg A$ as a notational definition for $A \supset \bot$:

$$\frac{A\,\mathsf{false}}{\neg A\,\mathsf{true}}\,\neg I \qquad \frac{\neg A\,\mathsf{true} \quad \begin{array}{c}\overline{A\,\mathsf{false}}^{\,k}\\ \vdots\\ J\end{array}}{J}\,\neg E^k$$

We wrote the rule $\neg E^k$ in the style of $\vee$-elim, but we could equivalently have given a rule that looks like $\wedge$-elim: from $\neg A\,\mathsf{true}$ conclude $A\,\mathsf{false}$. These two rules are equivalent in our setting here (though keep this distinction in mind when we talk about focusing in a couple of weeks).

1.1.3 Classical logic

Thus far, all of our new judgments are OK constructively. One way to see this to prove that the new judgements can be eliminated, treating $\#$ as a notational definition for $\bot\,\mathsf{true}$ and $A\,\mathsf{false}$ as $A \supset \bot\,\mathsf{true}$, and checking that all the new rules are derivable. This shows that we have not fundamentally changed the meaning of truth: the truth of $\neg A$, a new proposition, depends on contradiction and falsehood, but the truth of existing propositions such as $A \vee B$ is unchanged.

To get classical logic, we add the judgmental version of DNE:

$$\dfrac{\begin{array}{c}\overline{A\,\mathsf{false}}\ k\\ \vdots\\ \#\end{array}}{A\,\mathsf{true}}\ DNE^k$$

This rule does change the meaning of truth for existing propositions, because the A in the conclusion might be, e.g., a disjunction. Note the symmetry between DNE and fI: classical logic is more symmetric than intuitionistic logic, where truth means something stronger than merely not being false.

Exercise. Give a derivation of $(\neg(A \wedge B)) \supset \neg A \vee \neg B\,\mathsf{true}$. In Homework 2, you saw that this De Morgan principle was not intuitionistically true, but it is classically true.

Here's a Tutch-like proof to guide you along:

```
[~(A & B) true
 [A & B false
  [~A | ~B false
   [A true
    [B true
     A & B true
     #]
    B false
    ~B true
    ~A | ~B true
    #]
   A false
   ~A true
   ~A | ~B true
   #]
  ~A | ~B true]
 ~A | ~B true]
~(A & B) => ~A | ~B true
```

2 What is the relationship with intuitionistic logic?

Let's write $\Gamma \vdash_c A\,\mathsf{true}$ for classical truth and $\Gamma \vdash_i A\,\mathsf{true}$ for intuitionistic truth, where we put a context Γ of hypotheses in the judgement form rather than using the two-dimensional notation for hypotheses.

It's easy to prove that:

> If $\Gamma \vdash_i A\,\mathsf{true}$ then $\Gamma \vdash_c A\,\mathsf{true}$.

This says that if an intuitionist asserts A, a classicist will believe him, interpreting A classically. Informally, the move from intuitionistic to classical logic consisted of adding new inference rules, so whatever is true intuitionistically must be true classically. This can be formalized as a proof by rule induction on $\Gamma \vdash_i A\,\mathsf{true}$.

Of course, the opposite entailment does not hold (take A to be the law of the excluded middle, or double-negation elimination). However, it is possible to *translate* propositions in such a way that, if a proposition is classically true, then its translation is intuitionistically true. That is, the intuitionist does not believe what the classicist says at face value, but he can figure out what the classicist really meant to say, by means of a *double-negation translation*. The translation inserts enough double-negations into the proposition A that the classical uses of the DNE rule are intuitionistically permissible.

We will use the "Gödel-Gentzen negative translation", which is defined by a function $A^* = A'$ from classical propositions to intuitionistic propositions. On the intuitionistic side, we use the usual notational definition of

$\neg A = (A \supset \bot)$.

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

That is, the classicist and the intuitionistic agree about the meaning of all of the connectives except $\vee$ and atomic propositions P. From an intuitionistic point of view, when a classicist says $A \vee B$, he *really* means $\neg\neg(A^* \vee B^*)$, an intuitionistically weaker statement. Thus, **intuitionistic logic is more precise**, because you can say $A \vee B$, if that's the case, or $\neg\neg(A \vee B)$ if you need classical reasoning to do the proof. There is no way to express intuitionistic disjunction in classical logic. If an intuitionist says A to a classicist, and then the classicist repeats it back to him, it will come back as a weaker statement A^*.

On the other hand, the translation has the property that A and A^* are classically equivalent. If a classicist says something to an intuitionist, and then the intuitionist repeats it back to him, the classicist won't know the difference: intuitionistic logic makes finer distinctions.

As an aside, there are several other ways of translating classical logic into intuitionistic logic, which make different choices about where to insert double-negations. Different translations do different things to proofs, which turns out to have interesting consequences for programming.

How do we verify that this translation does the right thing? First, we need to lift the translation to judgments as follows:

$$\begin{array}{rcl}
(A\,\mathsf{true})^* & = & A^*\,\mathsf{true} \\
(A\,\mathsf{false})^* & = & (A \supset \bot)\,\mathsf{true} \\
(\#)^* & = & \bot\,\mathsf{true}
\end{array}$$

and to contexts Γ by translating each assumption in the context. Then we can prove:

Theorem 1. $\Gamma \vdash_c J$ *iff* $\Gamma^* \vdash_i J^*$.

Proof. The "only if" direction is a consequence of two lemmas mentioned above:

1. If $\Gamma \vdash_i J$ then $\Gamma \vdash_c J$
2. For all A, Γ, $\Gamma \vdash_c (A \supset A^*) \wedge (A^* \supset A)$ true.

The first is proved by rule induction on $\Gamma \vdash_i J$, the second by induction on A.

The "if" direction is proved by induction on $\Gamma \vdash_c J$. The hard case is eliminating uses of the DNE rule:

$$\frac{\Gamma, A\,\mathsf{false} \vdash_c \#}{\Gamma \vdash A\,\mathsf{true}}\ DNE$$

The inductive hypothesis is

$$(\Gamma, A\,\mathsf{false})^* \vdash_c (\#)^*$$

Expanding the definition of the translation gives:

$$\Gamma^*, (A^* \supset \bot)\,\mathsf{true} \vdash_c \bot\,\mathsf{true}$$

By implication introduction, this gives

$$\Gamma^* \vdash_i \neg\neg A^*$$

On the other hand, translating the conclusion of the rule, we need to show that

$$\Gamma^* \vdash_i A^*$$

.

Uh-oh! This is an instance of double-negation elimination, which we don't have intuitionistically! So why does the translation work?

The key is that we only need double-negation elimination *for the target of the translation*. That is, we need the translation to have the property that $\neg\neg(A^*) \supset A^*$. This lemma is true for the translation defined above (you can prove it by induction on A). But if, for example, we forgot to double-negate disjunction or atoms, the property would not be true. The lemma that

$$\neg\neg(A^*) \supset A^*$$

is how we tell that we've added enough double-negations to allow all the classical reasoning that we need.

□

3 Intermezzo: Truth tables

Once upon a time, someone told you that to check whether a proposition is (classically) true, you build a truth table. What are the reasoning principles behind truth tables?

1. To prove $P\ \mathsf{prop} \vdash A\ \mathsf{true}$, it suffices to prove $[\top/P]A\ \mathsf{true}$ and $[\bot/P]A\ \mathsf{true}$.
2. You compute the truth value of a connective from the truth values of its components, using equations like

$$\begin{array}{rcl} \top \supset \top & \equiv & \top \\ \top \supset \bot & \equiv & \bot \\ \bot \supset \top & \equiv & \top \\ \bot \supset \bot & \equiv & \bot \end{array}$$

The equations in (2) are true, constructively and classically, if you interpret $A \equiv B$ as $(A \supset B) \wedge (B \supset A)$.

But what about (1), the idea that to prove a proposition A, you can distinguish cases on the truth of an atomic proposition P appearing in A?. As you might expect, this reasoning is valid classically but not constructively. By the law of the excluded middle, we can case-analyze $P \vee \neg P$:

$$\dfrac{\dfrac{}{P \vee \neg P}\,LEM \qquad \begin{array}{c}\overline{P\ \mathsf{true}}^{\,u}\\ \vdots\\ J\end{array} \qquad \dfrac{\overline{\neg P\ \mathsf{true}}^{\,v} \quad \dfrac{\begin{array}{c}\overline{P\ \mathsf{false}}^{\,k}\\ \vdots\\ J\end{array}}{J}\,\neg E^{k}}{J}}{J}\,\vee E^{u,v}$$

Thus, to show J, it suffices to show $P\ \mathsf{true} \vdash J$ and $P\ \mathsf{false} \vdash J$.

So, to justify the truth-table reasoning, we just need the following lemma:

Lemma 2.

1. *For all J, if $[\top/P]J$ then $(P\ \mathsf{prop}, P\ \mathsf{true} \vdash J)$*
2. *For all J, if $[\bot/P]J$ then $(P\ \mathsf{prop}, P\ \mathsf{false} \vdash J)$*

Here's an intuition for why this is true: For the first part, everywhere the $\top I$ rule was used, we can instead use the assumption of $P\,\mathsf{true}$. For the second, everywhere the derivation uses $\bot E$ from a proof of $\bot\,\mathsf{true}$, we instead have a proof of $Ptrue$, which can be used with fE and $\#E$ to conclude anything.

4 What is the computational meaning of classical proofs?

4.1 Proof Terms

Let's annotate the above rules with proof terms:

$$\frac{M : \#}{\mathsf{abort}\,M : J} \qquad \frac{\begin{array}{c}\overline{u : A\,\mathsf{true}}\\ \vdots \\ M : \#\end{array}}{\mathsf{cont}\,u.M : A\,\mathsf{false}} \qquad \frac{M : A\,\mathsf{false} \quad N : A\,\mathsf{true}}{\mathsf{throw}\,M\,N : \#}$$

$$\frac{M : A\,\mathsf{false}}{\mathsf{not}\,M : \neg A\,\mathsf{true}} \qquad \frac{M : \neg A\,\mathsf{true} \quad \begin{array}{c}\overline{k : A\,\mathsf{false}}\\ \vdots \\ N : J\end{array}}{\mathsf{notcase}(M, k.N) : J}$$

$$\frac{\begin{array}{c}\overline{k : A\,\mathsf{false}}\\ \vdots \\ M : \#\end{array}}{\mathsf{letcc}(k.M) : A\,\mathsf{true}}$$

4.2 Programming with continuations

As you know, when you give a proof term assignment to some logical rules, the operator names (abort, throw, letcc, ...) are arbitrary. However, the names we've chosen here are fairly standard for the programming feature distinguishes classical proofs from constructive ones: *continuations*.

The term letcc(k.M) is short for "let the current continuation be k in M". What is the "current continuation"? It's all the work that's left to do in the rest of the program. In implementation terms, letcc gives the program access to its own control stack. In letcc, the current control stack gets packed up as value bound to k. Continuations are used by throwing them a value, which forgets about the current execution context and runs a stack that you previously saved on that value. Continuations can be thrown to multiple times, which makes the control flow in a program with continuations very different than the traditional push/pop behavior that you get from function calls in intuitionistic logic. You might save a stack, run it on a value, go off and do something else for a while, and then come back to that stack again with a different value. Unlike languages without letcc, control stacks must be implemented as persistent data structures, not just as an ephemeral piece of mutable memory. Continuations are a very general mechanism, and can be used to implement other control forms, such as exceptions, coroutines, and threads.

As an example of programming with continuations, consider a function that multiples all the integers in a list. In writing this code, we'll assume that `intlist` and `int` are *propositions*, like they would be in ML, and that we can write pattern-matching functions over them. Here's a first version:

```
mult' : intlist => int
mult' [] = 1
mult' (x :: xs) = x * mult' xs
```

I.e., the multiplication of the empty list is `1`, and the multiplication of `x :: xs` is the head times the multiplication of the tail.

What happens when we call `mult' [1,2,3,0,4,5,....]` where the `...` is 700 billion[1] more numbers? It does a lot more work than necessary to figure out that the answer is 0. Here's a better version:

```
mult' : intlist => int
mult' [] = 1
mult' (0 :: xs) = 0
mult' (x :: xs) = x * mult' xs
```

This version checks for 0, and returns 0 immediately, and therefore does better on the list `[1,2,3,0,4,5,....]`.

But what about the reverse list `[...,5,4,0,1,2,3]`? This version still does all 700 billion multiplications on the way up the call chain, which could also be skipped.

We can do this using continuations:

```
mult xs = letcc k : int false in
  let
```

[1]this week's trendy really-large number to pull out of thin air

```
   mult' : intlist => int
   mult' [] = 1
   mult' (0 :: xs) = abort(throw k 0)
   mult' (x :: xs) = x * (mult' xs)
  in throw k (mult' xs)
```

The idea is that we grab a continuation `k` standing for the evaluation context in which `mult` is called. Whenever we find a 0, we **immediately** jump back to this context, with no further multiplications. If we make it through the list without finding a zero, we throw the result of `mult'` to this continuation, returning it from `mult`.

Let's try to run `(mult [0,1,2]) + 5`. It's easiest to define evaluation for proofs of #, so we'll run this term against an initial continuation variable `halt : int false`.

```
        throw halt ((mult [0,1,2]) + 5)
  ⇒R   throw halt
         (letcc k in let ... in throw k (mult' [0,1,2])) + 5)
  ⇒R   (throw (cont u.throw halt (u + 5))
         (([(cont u.throw halt (u + 5))/k]mult') [0,1,2])
  ⇒R   (throw (cont u.throw halt (u + 5))
         (abort (throw (cont u.throw halt (u + 5)) 0)))
  ⇒R   (throw halt (0 + 5))
  ⇒R   (throw halt 5)
```

Some things to note:

- In the second reduction step, the whole expression enclosing the `letcc` is packed up as a `cont` and substituted for `k`.
- In the third reduction, we evaluate a `throw` by evaluating the proof of `A true`. Another choice would be to evaluate the proof of `A false`; these correspond to different *evaluation orders* for the programming language.
- In the fourth reduction, we have a `throw` whose true expression is fully evaluated, so we substitute this value into the continuation and forget the enclosing context.

What is the continuation assumption `halt`? This represents the initial continuation of the program (in practice, this might print the final value out to the user). We need this initial continuation assumption because there are no closed contradictions in classical logic!

Exercise. Recast the code for `mult` as a proof of $\forall x : intlist. \exists y : int. \top$, so `intlist` and `int` are treated as types of objects rather than as propositions.

As another example, returning to the proof of LEM above, we can now write it as:

```
letcc k : ~(A | ~A) in
 throw k (inr (not (cont v => k (inl v))))
```

Executing this code will resume the continuation `k` twice: first with `inr M`, and then, if `k` ever uses `M`, with `inl v` where `v` is the value that `k` supplies. The program "time travels" between different moments in its execution.

Exercise. Give proof terms for the following:

- $(A \supset B) \supset (\neg B \supset \neg A)$
- $(\neg B \supset \neg A) \supset (A \supset B)$

4.3 Continuation-Passing Style

Classical logic is a functional-programming language with letcc, and intuitionistic logic is a functional programming language without it. So what is the computational meaning of the double-negation translation A^*, which transforms classical forms into intuitionistic proofs? It is a transformation on programs that eliminates all uses of letcc, usually called a *continuation-passing style (CPS) translation*.

For example:

$$\begin{aligned}
(\mathsf{intlist} \supset \mathsf{int})^* &= \mathsf{intlist}^* \supset \mathsf{int}^* \\
&= \neg\neg\mathsf{intlist} \supset \neg\neg\mathsf{int} \\
&= \neg\neg\mathsf{intlist} \supset \neg\mathsf{int} \supset \bot
\end{aligned}$$

Here, a function that takes an intlist and returns an int is transformed into a function that:

1. takes an extra argument of type $\neg$int, representing the current continuation (hence, continuation-passing style)
2. never returns (because its result type is $\bot$)

CPS translation is used in compilers for several reasons: First, it reduces the problem of implementing a language with letcc to that of implementing a language without it. Second, even if you're compiling an intuitionistic language, there are reasons to CPS convert it: (1) The control flow that you have to implement is simpler, because functions call but never return. Consequently, there is no need for a control stack—or, more precisely, the control stack is represented as heap objects, just like all other values in the program. (2) CPS conversion makes certain optimizations, such as one called *tail-call optimization*, easier to do.

4.4 Running a CPS program

The CPS conversion of a program like `mult [1,2,3,4,5]` has type $\neg\mathsf{int} \supset \bot$. So how do you actually see what number it produced? One option is to extend the language with an initial continuation, as discussed above. Another is to revise the double-negation translation so use an arbitrary *answer type* in place of $\bot$. Let's define $\neg_\alpha A = A \supset \alpha$. Then the following double-negation translation works:

$$\begin{array}{rcl}
(\top)^* & = & \top \\
(\bot)^* & = & \alpha \\
(A \wedge B)^* & = & A^* \wedge B^* \\
(A \vee B)^* & = & \neg_\alpha\neg_\alpha(A^* \vee B^*) \\
(A \supset B)^* & = & (A^* \supset B^*) \\
(\neg A)^* & = & \neg_\alpha A^* \\
(P)^* & = & \neg_\alpha\neg_\alpha P
\end{array}$$

The proof is similar to above, but requires the additional lemma that for all A, $\alpha \supset A^*$.

For $\alpha = \bot$, the translation is the same as above. But what if we take $\alpha = \mathsf{int}$. Then

$$\begin{array}{rcl}
\mathsf{int}^* & = & \neg_{\mathsf{int}}\neg_{\mathsf{int}}\mathsf{int} \\
 & = & (\mathsf{int} \supset \mathsf{int}) \supset \mathsf{int}
\end{array}$$

Now we can supply the identity function `fn x => x` as the initial continuation, and get back an actual number.

Logically, this says that a classical proof of a base type P (from no hypotheses) determines an intuitionistic proof of P itself—*not* just the double-negation of P. The same device can be used to study other classes of propositions for which intuitionistic and classical logic agree.