Lecture Notes on Classical Computation

15-317: Constructive Logic
Ronald Garcia

Lecture 8
September 24, 2015

1 Introduction

In the last lecture, we talked about how to alter our system of logic to support classical reasoning. To do so, we introduced two new judgments: # signifying contradiction, and *A false*. In this lecture, we explore a computational interpretation of this new system.

We explore the correspondence between proofs in classical logic and programs. Adding contradiction and proof by contradiction to our logic leads to sophisticated control operators in our programs. We also discuss the relationship between classical logic and intuitionistic logic.

2 Proof Terms

We begin by associating proof terms with each of our new rules. The rule of contradiction follows:

$$\frac{k : A\ \mathit{false} \quad M : A\ \mathit{true}}{\mathbf{throw}^J\ k\ M : J}\ \mathit{contra}$$

Since contradiction can produce any judgment whatsoever, we annotate the proof term constructor as $\mathbf{throw}^J$ so that it precisely captures the proof. We will omit this annotation when it's clear from the context. Recall that *false* judgments are never introduced but only used as assumptions, hence the only proof term that the left premise can have is of the form k. Furthermore, since we now handle multiple judgments, we need to contrast A *true* from

A *false* and # also in the proof term assignment. Thus, proof term judgments are now of the form $M : J$ with a judgment J instead of just $M : A$ with a proposition A.

The proof term assignment for proof by contradiction is as follows:

$$\dfrac{\begin{array}{c}\overline{k : A\ \mathit{false}}\ k\\ \vdots\\ E : \#\end{array}}{\mathcal{C}k{:}A.\ E : A\ \mathit{true}}\ PBC^k$$

As with proof terms for implication introduction, we annotate the proof term variable as $\mathcal{C}k{:}A.E$ to disambiguate the proof.

We will study these two rules and their associated proof terms to see how classical logic corresponds to computation.

As you know from last time, $A \vee \neg A$, often called the law of the excluded middle (LEM), is true in classical logic but not in intuitionistic logic. Here is a proof of LEM annotated with terms:

$$\dfrac{\overline{k : A \vee \neg A\ \mathit{false}}\ k \qquad \dfrac{\dfrac{\dfrac{\overline{k : A \vee \neg A\ \mathit{false}}\ k \qquad \dfrac{\overline{v : A\ \mathit{true}}\ v}{\mathbf{inl}\ v : A \vee \neg A\ \mathit{true}}\ \vee I}{\mathbf{throw}\ k\ (\mathbf{inl}\ v) : \bot\ \mathit{true}}\ \mathit{contra}}{\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v) : \neg A\ \mathit{true}}\ \supset I^v}{\mathbf{inr}\ (\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v)) : A \vee \neg A\ \mathit{true}}\ \vee I}{\dfrac{\mathbf{throw}\ k\ (\mathbf{inr}\ (\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v))) : \#}{\mathcal{C}k.\mathbf{throw}\ k\ (\mathbf{inr}\ (\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v))) : A \vee \neg A\ \mathit{true}}\ PBC^k}\ \mathit{contra}$$

The proof term assignment for $\neg A$ agrees exactly with that of $A \supset \bot$. To simplify, this lecture once again uses $\neg A$ as notation for $A \supset \bot$. Observe that v is a proof of A *true*, while $\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v)$ is a proof of $\neg A$ *true*. Notice as well that **throw** is used to produce two separate judgments. Once it yields $\bot\ \mathit{true}$, which we need to produce a proof of $\neg A\ \mathit{true}$, and once it yields a contradiction #, which is used in a proof by contradiction. Finally, notice that here we're using *judgments as types* not just propositions as types. Since our proof terms represent more judgments than A *true*, it's not sufficient to simply give A as the type: we must capture whether A is true or false, or if we have produced a contradiction (#).

$$\dfrac{\dfrac{\dfrac{}{LEM : A \lor \neg A} \qquad \dfrac{\dfrac{\dfrac{}{A \supset B \lor C}\,f \quad \dfrac{}{A}\,x}{B \lor C}\supset E \quad \dfrac{\dfrac{\dfrac{}{B}\,u}{A \supset B}\supset I\text{-}}{(A \supset B) \lor (A \supset C)}\lor I \quad \dfrac{\dfrac{\dfrac{}{C}\,w}{A \supset C}\supset I\text{-}}{(A \supset B) \lor (A \supset C)}\lor I}{(A \supset B) \lor (A \supset C)}\lor E^{u,w} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A}\,y \quad \dfrac{}{A}\,z}{\bot}\supset E}{B}\bot E}{A \supset B}\supset I^{z}}{(A \supset B) \lor (A \supset C)}\lor I}{(A \supset B) \lor (A \supset C)}\lor E^{x,y}}{(A \supset B \lor C) \supset (A \supset B) \lor (A \supset C)}\supset I^{f}$$

Figure 1: Classical proof using the law of the excluded middle.

Figure 1 proves that $(A \supset B \lor C) \supset (A \supset B) \lor (A \supset C)$ using the law of excluded middle we just proved as a lemma:

$$LEM = \mathcal{C}k{:}A \lor \neg A.\mathbf{throw}^{\#}\ k\ (\mathbf{inr}^{A}\ (\lambda v{:}A.\mathbf{throw}^{\bot}\ k\ (\mathbf{inl}^{\neg A}\ v)))$$

The proof term that corresponds to this proof is as follows:

$$\begin{array}{l} \lambda f.\mathbf{case}\ LEM\ \mathbf{of} \\ \qquad \mathbf{inl}\ x \Rightarrow \mathbf{case}\ f\ x\ \mathbf{of} \\ \qquad\qquad\qquad \mathbf{inl}\ u \Rightarrow \mathbf{inl}\ (\lambda\_.u) \\ \qquad\qquad\qquad \mathbf{inr}\ w \Rightarrow \mathbf{inr}\ (\lambda\_.w) \\ \qquad \mathbf{inr}\ y \Rightarrow \mathbf{inl}\ (\lambda z.\mathbf{abort}(y\ z)) \end{array}$$

For a final example, consider Peirce's Law, $((A \supset B) \supset A) \supset A$.
In a previous class, we tried to write a function with this type, but found that it was not possible because it's not intuitionistically true. Now using classical logic, we can both prove it and program it!

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ \textit{false}}\,k \qquad \dfrac{\dfrac{}{(A \supset B) \supset A\ \textit{true}}\,f \qquad \dfrac{\dfrac{\dfrac{}{A\ \textit{false}}\,k \quad \dfrac{}{A\ \textit{true}}\,u}{\boxed{B}}\,\textit{contra}}{A \supset B\ \textit{true}}\supset I^{u}}{A\ \textit{true}}\supset E}{\#}\,\textit{contra}}{A\ \textit{true}}\,PBC^{k}}{((A \supset B) \supset A) \supset A\ \textit{true}}\supset I^{f}$$

Observe the boxed B. The contradiction rule concludes B *true* out of thin air (since contradictions were allowed to conclude any judgment) and uses

it to conclude that $A \supset B$ *true* thereby discharging our assumption of A *true*. We can then use this $A \supset B$ *true* to eliminate
$((A \supset B) \supset A) \supset A$ *true* and conclude A without ever having a real proof of A! Then we use proof by contradiction to discharge our assumption that A *false*. The proof term for Pierce's law is:

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

Notice that it **throw**s k twice: once to produce a type that we need out of thin air, (via contradiction) and once to judge # so that the variable k can be bound using $\mathcal{C}$. The proof of LEM does the same thing. This is a common pattern for proofs (and programs) that depend on classical logic.

3 Reduction

Adding proof by contradiction to our logic changes the meaning of all of our connectives because there are now new ways to introduce each of them: the introduction rules are no longer the sole means.

Having changed our system, it's now necessary to check that local soundness and local completeness still hold for each connective. Local completeness still holds exactly the same way: given a proof of say $A \wedge B$, one can still use the original elimination and introduction rules to perform a local expansion.

On the other hand, local soundness must be shown to apply for every combination of introduction and elimination rules. In our new system, proof by contradiction can be used to introduce every propositional connective, so we must show that proof-by-contradiction followed by an elimination can be reduced.

Looking at the rules of the system, it's not necessarily obvious how one could reduce a proposition introduced by contradiction. Let's demonstrate how using implication as an example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset B\ \mathit{false}}\,k \qquad \begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array}}{\#}\,\mathit{contra}}{A \supset B\ \mathit{true}}\,PBC^k \qquad \begin{array}{c}\mathcal{E}\\ A\ \mathit{true}\end{array}}{B\ \mathit{true}}\,\supset E \qquad \Longrightarrow_R$$

$$\dfrac{\dfrac{\dfrac{}{B\ \mathit{false}}\,k' \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array} \quad \begin{array}{c}\mathcal{E}\\ A\ \mathit{true}\end{array}}{B\ \mathit{true}}\,\supset E}{\#}\,\mathit{contra}}{B\ \mathit{true}}\,PBC^{k'}$$

Observe how the local reduction behaves. The PBC rule introduced an implication $A \supset B$ *true*, which was immediately eliminated. To reduce this rule, the elimination rule is pushed up to the point in the proof where the contra rule is applied to the assumption $A \supset B$ *false* which was labeled k. Furthermore, the contra rule is now applied to an assumption B *false* which is now labeled with k', and the PBC rule now discharges this new assumption (the variable name doesn't have to change, but it can; from here on we often keep it the same).

Referring to the corresponding proof terms, the reduction looks like the following:

$$(\mathcal{C}k.\mathbf{throw}\ k\ M)\ N \Longrightarrow_R \mathcal{C}k.\mathbf{throw}\ k\ (M\ N)$$

Here, M is the proof term for $\mathcal{D}$ and N is the proof term for $\mathcal{E}$.

Local reduction operates similarly in the case of conjunction:

$$\mathbf{snd}\ (\mathcal{C}k.\mathbf{throw}\ k\ M) \Longrightarrow_R \mathcal{C}k.\mathbf{throw}\ k\ (\mathbf{snd}\ M)$$

In this case, M is a proof term for $A \wedge B$ *true*, and the entire term is a proof of B *true*.

The two examples above can be generalized to describe how local reductions involving $\mathcal{C}k.E$ behave. Each elimination rule is associated with some operation: **fst** and **snd** for conjunction, **case** for disjunction, and **abort** for false. Whenever one of these operations is applied to a proof by contradiction, the reduction rule "steals" the operation and copies it to every location where the abstracted variable k is thrown in its body. For example:

$$\mathbf{snd}\ (\mathcal{C}k.E) \Longrightarrow_R \mathcal{C}k'.[k'\ (\mathbf{snd}\ \square)/k]E$$

The syntax $[k'\ (\textbf{snd}\ \Box)/k]E$ stands for a new kind of substitution, *structural substitution* into the body of E. Roughly speaking, in the body of E, every instance of **throw** $k\ M$ is replaced with **throw** $k'\ (\textbf{snd}\ M)$. 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.

Continuing the above reductions, the full set of rules are as follows:

$$\begin{array}{lll}
(\mathcal{C}k.\textbf{throw}\ k\ M)\ N & \Longrightarrow_R & \mathcal{C}k.\textbf{throw}\ k\ (M\ N) \\
\textbf{fst}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{fst}\ \Box)/k]E \\
\textbf{snd}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{snd}\ \Box)/k]E \\
\begin{array}{l}\textbf{case}\ (\mathcal{C}k.E)\ \textbf{of} \\ \quad \textbf{inl}\ u \Rightarrow M \\ \quad \textbf{inr}\ v \Rightarrow N\end{array} & \Longrightarrow_R & \mathcal{C}k'.\left[k' \begin{array}{l}\textbf{case}\ \Box\ \textbf{of} \\ \textbf{inl}\ u \Rightarrow M \\ \textbf{inr}\ v \Rightarrow N\end{array} \Big/ k\right] E \\
\textbf{abort}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{abort}\ \Box)/k]E
\end{array}$$

Not only can proof-by-contradiction introduce any logical connective, but plain ole' contradiction, which we associate with the **throw** operator, can as well! As we saw earlier, this was critical in proving Peirce's law and the law of the excluded middle. It turns out that a contradiction followed by an elimination can also be reduced. Here is an example using implication:

$$\dfrac{\dfrac{\dfrac{}{C\ \mathit{false}}\,k \qquad \begin{array}{c}\mathcal{D}\\ C\mathit{true}\end{array}}{A \supset B\ \mathit{true}}\,\mathit{contra} \qquad \begin{array}{c}\mathcal{E}\\ A\ \mathit{true}\end{array}}{B\ \mathit{true}}\supset E \qquad \Longrightarrow_R$$

$$\dfrac{\dfrac{}{C\ \mathit{false}}\,k \qquad \begin{array}{c}\mathcal{D}\\ C\mathit{true}\end{array}}{B\ \mathit{true}}\,\mathit{contra}$$

In this case, the local reduction *annihilates* the elimination rule, and now the *contra* rule simply concludes what the elimination rule used to. Here is the reduction written using proof terms:

$$(\textbf{throw}^{A\supset B}\ k\ M)\ N \Longrightarrow_R (\textbf{throw}^{B}\ k\ M)$$

Just like for $\mathcal{C}$, this reduction generalizes to the other elimination rules:

$$\begin{array}{lll}
\textbf{fst}\ (\textbf{throw}^{A\wedge B}\ k\ M) & \Longrightarrow_R & \textbf{throw}^{A}\ k\ M \\
\textbf{snd}\ (\textbf{throw}^{A\wedge B}\ k\ M) & \Longrightarrow_R & \textbf{throw}^{B}\ k\ M \\
\textbf{case}\ (\textbf{throw}^{A\vee B}\ k\ M)\ \textbf{of} & & \\
\quad \textbf{inl}\ u \Rightarrow M & \Longrightarrow_R & \textbf{throw}^{C}\ k\ M \\
\quad \textbf{inr}\ v \Rightarrow N & & \\
\textbf{abort}^{C}\ (\textbf{throw}^{\perp}\ k\ M) & \Longrightarrow_R & \textbf{throw}^{C}\ k\ M
\end{array}$$

In each case, **throw** eats any attempt to eliminate it, and like a chameleon, dresses itself up to look like what it should have been eliminated into! In the case of disjunction elimination, C is the proposition that is proved by both M and N.

To see these reductions in action, let's revisit our proof that $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$. To keep things somewhat manageable, we will only consider reductions on its proof term which is:

$$\begin{array}{l}
\lambda f.\textbf{case}\ \mathit{LEM}\ \textbf{of} \\
\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))
\end{array}$$

where

$$\mathit{LEM} = (\mathcal{C}k.\textbf{throw}\ k\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v))))$$

We can perform local reductions wherever a rule applies. We'll try to be systematic, working our way from the outside in. First, the **case** operator is eliminating the $\mathcal{C}$ inside of LEM, which introduces $A \vee \neg A$, so we can use our "steal and copy" reduction rule:

$$\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))))) \\
\qquad\qquad\qquad \textbf{of} \\
\qquad\qquad\qquad\quad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad\qquad\qquad\quad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z)))
\end{array}$$

The **case** operation has been absorbed into the $\mathcal{C}$ operator, and wrapped around the second argument to both throw expressions.

Now, the second argument to the first **throw** is a **case** applied to an **inr**. Recall that before this **case** was "stolen", it was eliminating the law of the excluded middle, $A \vee \neg A$, and here it finds out that our evidence is *really* (wink wink) a proof of $\neg A$, since it's wrapped in an **inr**. We can reduce this disjunction introduction (**inr**) and elimination (**case**) by substituting the content of the **inr** into the corresponding branch of the **case** expression:

$$
\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{abort}((\lambda v.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z)))))\ z))
\end{array}
$$

Next, in the argument to **abort**, $\lambda v...$ is applied to z so we can substitute z for v:

$$
\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{abort}(\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ z)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))))))
\end{array}
$$

Now there is an **abort** applied to a **throw**. According to our new local reductions, **throw** can eat **abort** (how tragic!):

$$
\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ z)\ \textbf{of} \\
\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\
\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))))
\end{array}
$$

Now we're back at the *same* **case** statement that we started with! It seems like we've gone back in time and have to resolve this case all over again, but this time the **case** expression is eliminating an **inl**, i.e. a proof of A! We can reduce it using the same strategy, but substituting into the other branch of the case statement:

$$\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{throw}\ k\ \textbf{case}\ f\ z\ \textbf{of} \\
\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\
\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w)
\end{array}$$

At this point, we've performed all of the local reductions that we can. What are we left with? Figure 2 is the proof tree corresponding to this term. It's a proof of the same theorem, $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$ *true*, but this one uses proof by contradiction directly instead of assuming the law of the excluded middle.

$$\begin{prooftree}
\AxiomC{}
\RightLabel{k}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *false*}
\AxiomC{}
\RightLabel{k}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *false*}
\AxiomC{}
\RightLabel{f}
\UnaryInfC{$A \supset B \lor C$ *true*}
\AxiomC{}
\RightLabel{z}
\UnaryInfC{A *true*}
\RightLabel{$\supset E$}
\BinaryInfC{$B \lor C$ *true*}
\AxiomC{}
\RightLabel{u}
\UnaryInfC{B *true*}
\RightLabel{$\supset I-$}
\UnaryInfC{$(A \supset B)$ *true*}
\RightLabel{$\lor I$}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *true*}
\AxiomC{}
\RightLabel{w}
\UnaryInfC{C *true*}
\RightLabel{$\supset I-$}
\UnaryInfC{$(A \supset C)$ *true*}
\RightLabel{$\lor I$}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *true*}
\RightLabel{$\lor E^{u,w}$}
\TrinaryInfC{$(A \supset B) \lor (A \supset C)$ *true*}
\RightLabel{*contra*}
\BinaryInfC{B *true*}
\RightLabel{$\supset I^z$}
\UnaryInfC{$A \supset B$ *true*}
\RightLabel{$\lor I$}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *true*}
\RightLabel{*contra*}
\BinaryInfC{#}
\RightLabel{PBC^k}
\UnaryInfC{$(A \supset B) \lor (A \supset C)$ *true*}
\RightLabel{$\supset I^f$}
\UnaryInfC{$(A \supset B \lor C) \supset (A \supset B) \lor (A \supset C)$ *true*}
\end{prooftree}$$

Figure 2: Direct proof of $(A \supset B \lor C) \supset (A \supset B) \lor (A \supset C)$ *true*

Looking at how we arrived at this proof, you can see how the classical operators:

1. lie about what they are proofs of;
2. steal any attempt to apply elimination rules to them; and
3. send you through time warps so you can repeat the same work but make different choices each time.

4 Relating Classical Logic to Intuitionistic Logic

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

It's easy to prove that:

If $\Gamma \vdash_i A$ *true* then $\Gamma \vdash_c A$ *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 $\Gamma \vdash_i A$ *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 $\neg A = (A \supset \bot)$.

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

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

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.

The following statement captures what we mean when we say that this translation "does the right thing".

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

5 Programming with Classical Logic: Continuations

Earlier we showed the proof term for Peirce's Law:

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

This proof of $((A \supset B) \supset A) \supset A$ corresponds to a powerful programming language construct. Peirce's law is the type of "call with current continuation" or `callcc`, a powerful operator that appears in the Scheme programming language and in Standard ML of New Jersey. Thinking operationally, `callcc` gives you a copy of the current call stack of a program that you can hold on to. Then later, after the program has done some more work and the stack has changed, you can replace the current stack with your old stack, in a sense turning back the sands of time to an old program state.

The `callcc` operator has an immediately-binding variant called **letcc**:

$$\mathbf{letcc}(k.M) \equiv \mathcal{C}k.\mathbf{throw}\ k\ M$$

With `callcc` in Scheme, the **throw** operation is wrapped up inside of a function ($\lambda u.$**throw** $k\ u$), so you can just call the continuation like any other function. The **letcc** operator requires you to explicitly **throw** the continuation k, just like with the $\mathcal{C}$ operator. Notice that both `callcc` and **letcc** immediately throw the continuation that they capture.

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
```

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 in
  let
```

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

```
   mult' : intlist => int
   mult' [] = 1
   mult' (0 :: xs) = 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`. Note that we could have just returned this value since `letcc k` has a `throw k` built-in.

In this program, continuations have been used simply to jump to the top of a computation. Other more interesting programs use continuations to jump out and then back in to a computation, resuming where you left off.

6 Continuation-Passing Style: double-negation translation for Programs

Continuations are a powerful programming language feature for classical functional programs, but what if you only have an intuitionistic functional programming language? Well, recall that there are several kinds of double-negation transformations which can embed classical logic propositions into intuitionistic logic. It turns out that this same strategy can be applied to programs: we can translate classical programs into intuitionistic programs. The process is called a *continuation-passing style* transformation, or CPS for short, and just as there are several different double-negation translations, there are several different CPS's. The Gödel-Gentzen negative translation double-negates all atomic propositions, which under our computational interpretation would correspond to changing all base types. We use another translation instead.

For the moment, let's limit ourselves to a logic with only implication $\supset$ and falsehood $\bot$. Another translation $A^\star$ is defined as follows:

$$\begin{array}{lcl} P^\star & = & P \\ (A \supset B)^\star & = & (A^\star \supset \neg\neg B^\star) \end{array}$$

and has the property that A holds classically iff $\neg\neg A^\star$ intuitionistically.

A corresponding program translation $\overline{M}$ takes a program M of type A to a program $\overline{M}$ of type $(A^\star \supset \bot) \supset \bot$:

$$\begin{array}{lcl}
\overline{x} & = & \lambda k.k\ x \\
\overline{\lambda x.M} & = & \lambda k.k\ (\lambda x.\overline{M}) \\
\overline{M\ N} & = & \lambda k.\overline{M}(\lambda f.\overline{N}(\lambda x.f\ x\ k) \\
\overline{\textbf{letcc}\ k_0.M} & = & (\lambda k.\overline{M}\ k)[(\lambda a.\lambda k.k\ a)/k_0] \\
\overline{\textbf{throw}\ k_0\ M} & = & \overline{k_0\ M}
\end{array}$$

Looking at the types yields, e.g.:

$$\begin{array}{lll}
\text{If } x : A & \text{then} & \overline{x} : (A^\star \supset \bot) \supset \bot \\
\text{If } \lambda x.\ M : A \supset B & \text{then} & \overline{x} : (\Big(A^\star \supset ((B^* \supset \bot) \supset \bot)\Big) \supset \bot) \supset \bot \\
\text{If } M\ N : B & \text{then} & \overline{M\ N} : (B^\star \supset \bot) \supset \bot \\
\text{where } M : A \supset B \text{ and } N : A & \text{so} & \overline{M} : (\Big(A^\star \supset ((B^* \supset \bot) \supset \bot)\Big) \supset \bot) \supset \bot \\
 & \text{and} & \overline{N} : (A^\star \supset \bot) \supset \bot
\end{array}$$

The computational way to read the resulting typing judgment $\overline{M} : (A^\star \supset \bot) \supset \bot$ is that if M was a program producing values of type A, then $\overline{M}$ is a program that gives you result values (of type $\bot$ here) directly if only you first hand a continuation of type $A^\star \supset \bot$ to $\overline{M}$ that can compute final results directly from the value A that M evaluated to. Except that the CPS transformation is applied recursively so that continuation-aware A^* instead of A will be passed into the continuation to compute the final result. For $\overline{M\ N}$, for example, $\overline{M}$ will compute a function so can be continued with λf to remember the function and calling $\overline{N}$ which will be continued with a continuation that accepts its resulting value by λx and finally computes the overall outcome fx except that that computation also will be passed in the original continuation k passed in to $\overline{M\ N}$ in the first place.

So CPS will turn a program of type `intlist -> int` into a program of type `(intlist -> (int -> 'o) -> 'o) -> 'o -> 'o` where `'o` is some type that plays the role of $\bot$. As it turns out, `'o` ends up being the type of the whole program.

To give some flavor for this translation, let's look at our mult function after CPS:

```
mult-k : intlist -> (int -> 'o) -> 'o
mult-k xs k0 =
  let k = k0 in
  let
   mult-k' : intlist -> (int -> int) -> int
   mult-k' [] k1 = k1 1
   mult-k' (0 :: xs) k1 = k 0
   mult-k' (x :: xs) k1 = (mult-k' xs (fn v => k1 (x * v)))
  in (mult-k' xs k)

mult-cps : (intlist -> (int -> 'o) -> 'o) -> 'o -> 'o
mult-cps = fn k => k mult
```

The functions above are a simplified version of the output of CPS, so there are not as many `fn k =>...` functions as would come out of the literal translation. The function `mult-cps` is the CPS counterpart of the original `mult` function.

To simplify matters, we'll work directly with `mult-k`. In contrast to the original `mult` function, this one takes an extra argument, `k0` of type `int -> 'o`. This is a function-based representation of the current continuation, all of the work that's left to be done. The first line in the function stores a copy of this continuation in a variable `k`. This is the counterpart to `letcc` from the classical program.

Now instead of throwing k, we can simply call the function `k` and pass it a value. Notice how everywhere in `mult-k'` that used to simply return a value, it now passes that value to the current continuation. Also, where it used to say `x * (mult' xs)` now calls the function `mult-k'` and passes it a *bigger* continuation that accepts a value `v`, the result of `mult-k' xs`, multiplies that value by `x`, then calls the continuation `k1`. This is how the stack grows. Any time there used to be more work to do after a function call returns, it has now been rolled into the continuation that is passed to that function.

Finally notice that the `0` case of `mult-k'` ignores its current continuation `k1` and instead calls the continuation that was captured at the top of `mult-k`. So how do we get a value back from this program? Well, we can pass it a continuation that simply returns whatever it gets:

```
id : int -> int
id x = x

mult-k big-list id
```

Now we've substituted the type `int` for the indeterminate type `'o`, and when the computation is completed, it will call this initial continuation `id` which returns the answer.

The CPS transformation is not simply an academic exercise. Some compilers use CPS internally while translating programs into executables. This makes it really easy to provide support for language features like **letcc** and similar operators.