Constructive Logic (15-317), Fall 2015
Recitation 7: Cut Elimination, Dependent Types (10/14/2015)
Vincent Huang (vincenth@andrew), Anna Gommerstadt (hgommers@cs), Michael Coblenz (mcoblenz@cs)

1 A quick refresher: weakening

Suppose $\Gamma, A \Longrightarrow B$. Give an example of a *weaker* statement.

One answer: $\Gamma, A, C \Longrightarrow B$ is weaker because it assumes an additional hypothesis.

Now give an example of a *stronger* statement.

$\Gamma \Longrightarrow B$ is stronger because it no longer needs a hypothesis.

So, remember: weakening adds hypothesis. We frequently write proofs bottom-up (backward!), so you might get confused and think that weakening removes hypotheses, but that is itself backward.

2 Cut Elimination

Cut elimination is a crucial aspect of logical systems; with cut elimination, we can exploit divide-and-conquer to write our proofs. But as you might have surmised from the complexity of the cut elimination proof, cut elimination is not an obvious result.

You might not yet be comfortable with the idea of inductive proofs that do induction on more than one structure at a time. You already know that induction works by defining a size metric so that the induction hypothesis applies to anything "smaller" than the current case. Usually "smaller" is obvious: a lesser natural number, for example. With multidimensional induction, we're doing induction on multiple structures simultaneously and carefully defining "smaller" so that (a) the thing we need to apply the induction hypothesis to happens to always be smaller (how convenient!) but also so that (b) we can show that if the structure keeps getting smaller, it eventually gets down to some kind of empty case (i.e. there are no cycles in our proof). Note that in structural induction, we don't need a base case because one or more of the ways of building the structure serve that purpose.

Today, we'll cover one more case in the cut elimination proof: disjunction.

Recall that we are in this situation:

$$\Gamma \overset{\mathcal{D}}{\Longrightarrow} A \qquad \Gamma, A \overset{\mathcal{E}}{\Longrightarrow} C$$

and we need to show that $\Gamma \overset{\mathcal{F}}{\Longrightarrow} C$. Our plan is to produce $\mathcal{F}$, which completes our proof.

Case: A is the principal formula of the final inference in both $\mathcal{D}$ and $\mathcal{E}$.

Subcase: If A is of the form $A_1 \vee A_2$, then

$$\mathcal{D} = \dfrac{\Gamma \overset{\mathcal{D}_1}{\Longrightarrow} A_1}{\Gamma \Longrightarrow A_1 \vee A_2} \vee R_1$$

and

$$\mathcal{E} = \dfrac{\Gamma, A_1 \vee A_2, A_1 \overset{\mathcal{E}_1}{\Longrightarrow} C \quad \Gamma, A_1 \vee A_2, A_2 \overset{\mathcal{E}_2}{\Longrightarrow} C}{\Gamma, A_1 \vee A_2 \Longrightarrow C} \vee L$$

Remember the form of the induction hypothesis: we can apply it to a term A, a proof $\mathcal{D}$ of $\Gamma \Longrightarrow A$, and a proof $\mathcal{E}$ of $\Gamma, A \Longrightarrow C$. We can apply the induction hypothesis with a strictly smaller cut formula (A), or with an identical cut formula and two derivations, one of which is strictly smaller while the other stays the same. Every use of the induction hypothesis has to have the right form (a term and two derivations) and satisfy the above size constraints so we make sure we don't induct in cycles.

Note that we're not going to need $\mathcal{E}_2$ because we know we're in the A_1 case, not the A_2 case and because we're not going to justify this with $\vee L$.

$\Gamma \Longrightarrow A_1 \vee A_2$	by $\mathcal{D}$
$\Gamma, A_1 \Longrightarrow C$	by induction hypothesis on $A_1 \vee A_2$, $\mathcal{D}$, and $\mathcal{E}_1$. Note $\mathcal{E}_1$ has a smaller derivation than $\mathcal{E}$ does. Also note we have to use weakening on $\mathcal{D}$ to add an extra hypothesis A_1, before we can appeal to the induction hypothesis on $\mathcal{E}_1$ because $\mathcal{E}_1$ has an extra hypothesis.
$\Gamma \Longrightarrow C$	by induction hypothesis on A_1, $\mathcal{D}_1$, and proof from previous line. Note A_1 is smaller than A.

3 Dependent Types

You're probably used to terms that depend on terms, such as functions, and maybe you're used to types that depend on types, as in parametric polymorphism. But what about types that depend on terms? We want to write functions that take *terms* as input and return types.

Recall the rules from lecture:

$$\frac{\Sigma, c : \tau; \Gamma \Longrightarrow A(c)}{\Sigma; \Gamma \Longrightarrow \forall x.A(x)} \forall R \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma, \forall x : \tau.A(x), A(t) \Longrightarrow C}{\Sigma; \Gamma, \forall x : \tau.A(x) \Longrightarrow C} \forall L$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma \Longrightarrow A(t)}{\Sigma; \Gamma \Longrightarrow \exists x : \tau.A(x)} \exists R \qquad \frac{\Sigma, c : \tau; \Gamma, \exists x : \tau.A(x), A(c) \Longrightarrow C}{\Sigma; \Gamma, \exists x : \tau.A(x) \Longrightarrow C} \exists L$$

Let's see how we might use these types (propositions) by starting without them. Let's suppose we have a type `Array` corresponding to arrays of natural numbers. Suppose we have a function f : `Array` $\rightarrow$ `Array` $\rightarrow$ `Array`. What do we know about what this function does? Are there some inputs that would typecheck but for which f might give runtime errors? Take a guess as to what f might do and give an example of an input that might give a runtime error:

Let's see if we can make the type more specific. Suppose `f` : Πn : `nat.Array`$(n) \supset$ `Array`$(n) \supset$ `Array`$(2 * n)$. What can you say about what f does now? What names might be more appropriate than f? Here we have some confusing terminology: perhaps the logic community prefers $\forall$ but the programming languages community prefers Π.

Now, suppose we want to keep track of the length of the list. We could use a pair of $\langle$`nat`, `Array`$\rangle$. But then the length might be wrong, since the programmer is responsible for keeping track of it! A better way is to seal the size inside the list with an existential type: Σn : `nat.Array`(n). Σ means the same thing as $\exists$. Now, look at the left and right rules. What do you expect to be able to do with terms of existential type?

This all seems great! But suppose you wanted to do this in a real programming language. What problems might you have, practically speaking?