Constructive Logic (15-317), Fall 2015
Assignment 6: Dependent Types Cut In!

Contact: Vincent Huang (vincenth@andrew.cmu.edu)
Recitation FILL ME IN

Due Tuesday, October 20, 2015

I thought this class was about logic! Will these types never end...

Ah, but you were tricked, you see. After all, they're the ~same thing~.

The written portion of your work (everything!) should be submitted at the beginning of class. If you are familiar with LaTeX, you are encouraged to use this document as a template for typesetting your solutions, but you may alternatively write your solutions *neatly* by hand.

1 Lists (7 points)

You will be using lists and primitive recursion on lists for 2 problems on the homework (including this one). Here are some of the relevant rules:

List formation

$$\frac{\tau\ \mathit{type}}{\tau\ \mathbf{list}\ \mathit{type}}\ \mathbf{list}F$$

List elimination

$$\frac{\Gamma \vdash t : \tau\ \mathbf{list} \qquad \Gamma \vdash s_n : \sigma \qquad \Gamma, x : \tau, l : \tau\ \mathbf{list}, r : \sigma \vdash s_c : \sigma}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ \mathbf{nil} \Rightarrow s_n \mid r, (x :: l) \Rightarrow s_c : \sigma}\ \mathbf{list}E$$

Task 1 (3 points)**.** Give the introduction rule(s) that correspond to this elimination rule.

Task 2 (4 points)**.** Give the proof term reduction(s) that correspond to the proof term assignment for $\mathbf{list}E$.

Side remark: this question goes against the "verificationist tradition", since we give the elimination rule and ask you for the introduction rule(s), rather than the other way round. Ah well. The Eliminati get around.

2 Mild Dependency (15 points)

Consider the replicate function *rep* that takes a natural number n and an arbitrary element a and returns a list of n copies of a:

$$rep \in \mathbf{nat} \to \tau \to \tau\ \mathbf{list}$$

$$rep\ 0\ a = \mathbf{nil}$$
$$rep\ (\mathbf{s}(n'))\ a = a :: (rep\ n'\ a)$$

Task 3 (3 points)**.** Give an explicit definition of *rep* by primitive recursion.

Now we would like to prove the basic assertion that *rep* n a is a list of length n.

Task 4 (2 points)**.** Give the type of *rep* using the dependent list type defined in lecture *so that we can use the previous specification unchanged.*

$$rep : \Pi n : \mathbf{nat}.\ \_\_\_\_\_$$

We now prove that the specification is well typed. We begin with the first line.

Task 5 (3 points)**.** The left hand side has type:

$$rep\ 0 : \_\_\_\_\_$$
$$rep\ 0\ a : \_\_\_\_\_$$

Task 6 (2 points)**.** The right hand side has type:

$$\mathbf{nil} : \_\_\_\_\_$$

Task 7 (5 points)**.** The left and right hand sides in the second line have type:

$$rep(\mathbf{s}(n')) : \_\_\_\_\_$$
$$rep(\mathbf{s}(n'))\ a : \_\_\_\_\_$$

$$rep\ n'\ a : \_\_\_\_\_$$
$$a :: (rep\ n'\ a) : \_\_\_\_\_$$

This shows that *rep* indeed does return a list of length n. Note that all the computation was done by the type checker, and that no runtime calculations were necessary!

3 Morbid Dependency (4 points)

Consider the function *zip*, probably familiar to you from 15-150, that takes 2 lists and pairs their elements together.

$$zip : \tau_1\ \mathbf{list} \to \tau_2\ \mathbf{list} \to (\tau_1 \times \tau_2)\ \mathbf{list}$$
$$zip : \tau_1\ \mathbf{list} \to \tau_2\ \mathbf{list} \to\ \mathbf{unit} + (\tau_1 \times \tau_2)\ \mathbf{list}$$

The first type signature corresponds to a definition of *zip* where the output list is as long as the length of the shorter list, and the second corresponds to a definition where *zip* fails if its two inputs are not exactly the same length (think of $\mathbf{unit} + \tau$ as an option type).

We wish to write dependently typed versions of this function. As we saw above with the non-dependently typed versions, there should be at least two ways of doing this.

Task 8 (2 points)**.** Give a dependent type for *zip* using only Π types (no code necessary, just the type).

Task 9 (2 points)**.** Give a dependent type for *zip* using Σ types, and Π types if necessary (no code necessary, just the type).

4 Cut elimination (10 points)

In lecture, we proved

Theorem (Cut)**.** If $\Gamma \Longrightarrow A$ and $\Gamma, A \Longrightarrow C$, then $\Gamma \Longrightarrow C$.

by nested induction first on the structure of A, then on the derivation $\mathcal{D}$ of $\Gamma \Longrightarrow A$, then on the derivation $\mathcal{E}$ of $\Gamma, A \Longrightarrow C$.

We also saw the sequent calculus extended to present quantification. As a quick reminder, for quantification, we add the judgement $a : \tau$, and collect all such hypotheses into a *signature* Σ. Sequents then have the form $\Sigma; \Gamma \Longrightarrow A$ where Σ is either empty or of the form $a_1 : \tau_1, \ldots, a_n : \tau_n$, and Γ is either empty or of the form $A_1, \ldots, A_n$. We also use the notation $\Sigma \vdash a : \tau$ to express that term a has type τ under signature Σ. Our cut theorem is then

Theorem (Cut)**.** If $\Sigma; \Gamma \Longrightarrow A$ and $\Sigma; \Gamma, A \Longrightarrow C$, then $\Sigma; \Gamma \Longrightarrow C$.

where in every case not involving quantifiers, the rules simply carry over the signature Σ without change.

In the remainder of this section, you may use the following 2 lemmas if you wish:

Lemma 1 (Substitution for parameters)**.** If $\Sigma \vdash t : \tau$ and $\Sigma, c : \tau; \Gamma \vdash A$, then

$$\Sigma; [t/c]\Gamma \vdash [t/c]A$$

Lemma 2 (Weakening for signatures)**.** If $\Sigma; \Gamma \Longrightarrow A$, then for fresh c, we have that

$$\Sigma, c : \tau; \Gamma \Longrightarrow A$$

with a structurally identical deduction.

Give proofs of the following cases which we did not do in lecture:

Task 10 (5 points)**.** The last rule applied in $\mathcal{E}$ is $\forall R$, and the principal formula in that application is not A.

Task 11 (5 points)**.** The last rule applied in $\mathcal{D}$ is $\exists L$, and the principal formula in that application is not A.

5 Cut it out!? (4 points)

We have proven cut elimination/admissibility in our system of sequent calculus, meaning we never *need* the cut rule in any of our derivations. However, eliminating cut can be annoying.

Task 12 (2 points)**.** Give a derivation of

$$B \wedge A \Longrightarrow ((A \wedge B) \vee C) \wedge (D \vee (A \wedge B))$$

that does *not* use the cut rule.

Task 13 (2 points)**.** Give a derivation of the same thing, but in such a way that your proof becomes smaller! (Think of it as using a lemma)