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

\[
\frac{\tau \text{ type}}{\tau \text{ list type} \quad \text{listF}}
\]

List elimination

\[
\frac{\Gamma \vdash t : \tau \text{ list} \quad \Gamma \vdash s_n : \sigma \quad \Gamma, x : \tau, l : \tau \text{ list}, r : \sigma \vdash s_c : \sigma}{\Gamma \vdash \text{rec } t \text{ of } \text{nil} \Rightarrow s_n \mid r, (x :: l) \Rightarrow s_c : \sigma \quad \text{listE}}
\]

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

Solution:

\[
\frac{\Gamma \vdash \text{nil} : \tau \text{ list}}{\text{listnil} \quad \text{nil}} \quad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash l : \tau \text{ list}}{\Gamma \vdash x :: l : \tau \text{ list} \quad \text{listcons}}
\]

Task 2 (4 points). Give the proof term reduction(s) that correspond to the proof term assignment for listE.

Solution:

\[
\text{rec nil of nil } \Rightarrow s_n \mid r, (x :: l) \Rightarrow s_c \Rightarrow_R s_n
\]

\[
\text{rec } h :: t \text{ of nil } \Rightarrow s_n \mid r, (x :: l) \Rightarrow s_c \Rightarrow_R [h/x][t/l] \quad [\text{rec } t \text{ of nil } \Rightarrow s_n \mid r, (x :: l) \Rightarrow s_c/r]s_c
\]
Some people used punning for the second reduction rule, i.e. called the list they were operating on \(x :: l\) to match up with the variables in the cons case of the rec. This is fine. The important thing to take away is \(s_c\) can mention all three of \(r, x\) and \(l\), and you have to provide them, using proper substitution notation.

Some people gave answers that substituted a rec for \(r(l)\) or something similar. While this is indeed what Pfenning did in his notes, notice that the typing context in his \(\text{list}E\) rule is slightly different from the one here, which is why this solution wasn’t given full credit.

# 2 Mild Dependency (15 points)

Consider the replicate function \(\text{rep}\) that takes a natural number \(n\) and an arbitrary element \(a\) and returns a list of \(n\) copies of \(a\):

\[
\text{rep} \in \mathit{nat} \to \tau \to \tau \mathit{list}
\]

\[
\text{rep} 0 a = \text{nil}
\]

\[
\text{rep} (s(n')) a = a :: (\text{rep} n' a)
\]

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

**Solution:**

\[
\text{rec} = \lambda \ n: \mathit{nat}. \ \lambda \ a: \tau. \ \text{R}(n, \ \text{nil}, \ x. \ r. \ \text{a} :: \ r)
\]

(You could also do things like put the \(\lambda a: \tau\) inside the \(\text{R}\). Make sure that, whatever you do, the types make sense! Also, don’t forget to do things like put \(\lambda s\) in front to bind the variables!)

Now we would like to prove the basic assertion that \(\text{rep} n a\) is a list of length \(n\).

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

\[
\text{rep} : \Pi n: \mathit{nat}. \ __________________________
\]
Solution:

\[ rep : \Pi n : \text{nat}. \ \tau \to \tau \text{list}(n) \]

People gave solutions that looked like

\[ rep : \Pi n : \text{nat}. \ \Pi \alpha : \tau. \ \tau \text{list}(n) \]

While this was accepted, note that it is unnecessarily dependent, since the rest of the type never mentions \( \alpha \) ... save such things for morbid dependency! :)

Common mistakes:

- having a \( \Pi n : \text{nat} \), then another \( \text{nat} \to \). The \( \Pi \) already takes in a \( \text{nat} \) argument! You shouldn’t be taking in another one!

- forgetting to write \( \text{list}(n) \) rather than just plain old \( \text{list} \).

- Using a \( \Sigma \) type here. This makes no sense. You want your \( rep \) function to operate on any natural number input...

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:

\[
\begin{align*}
\text{Task 5} &: \text{rep } 0 : \quad \text{__________________________} \\
\text{Task 5} &: \text{rep } 0 \ \alpha : \quad \text{__________________________}
\end{align*}
\]

Solution:

\[
\begin{align*}
\text{rep } 0 : \ \tau \to \tau \text{list}(0) \\
\text{rep } 0 \ \alpha : \ \tau \text{list}(0)
\end{align*}
\]

Some people wrote \( \alpha \text{list}(0) \) for the latter. \( \alpha \) is an element of type \( \tau \), not a type! Also, don’t forget the numbers; they’re kind of the entire point of this enterprise.

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

\[
\begin{align*}
\text{Task 6} &: \text{nil : } \quad \text{__________________________} \\
\end{align*}
\]

Solution:

\[
\begin{align*}
\text{nil} : \ \tau \text{list}(0)
\end{align*}
\]
Task 7 (5 points). The left and right hand sides in the second line have type:

\[
\begin{align*}
\text{rep(s(n'))} &: \quad \text{_______________________________} \\
\text{rep(s(n')) a} &: \quad \text{_______________________________} \\
\text{rep n’ a} &: \quad \text{_______________________________} \\
\text{a :: (rep n’ a)} &: \quad \text{_______________________________}
\end{align*}
\]

Solution:

\[
\begin{align*}
\text{rep(s(n'))} &: \tau \rightarrow \tau \; \text{list(s(n'))} \\
\text{rep(s(n')) a} &: \tau \; \text{list(s(n'))} \\
\text{rep n’ a} &: \tau \; \text{list(n')} \\
\text{a :: (rep n’ a)} &: \tau \; \text{list(s(n'))}
\end{align*}
\]

This shows that \text{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 \text{zip}, probably familiar to you from 15-150, that takes 2 lists and pairs their elements together.

\[
\text{zip} : \tau_1 \; \text{list} \rightarrow \tau_2 \; \text{list} \rightarrow (\tau_1 \times \tau_2) \; \text{list}
\]

\[
\text{zip} : \tau_1 \; \text{list} \rightarrow \tau_2 \; \text{list} \rightarrow \text{unit + (}\tau_1 \times \tau_2\text{)} \; \text{list}
\]

The first type signature corresponds to a definition of \text{zip} where the output list is as long as the length of the shorter list, and the second corresponds to a definition where \text{zip} fails if its two inputs are not exactly the same length (think of \text{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 \text{zip} using only \(\Pi\) types (no code necessary, just the type).

Solution:

\[
\Pi n : \text{nat.} \; \tau_1 \; \text{list(n)} \rightarrow \tau_2 \; \text{list(n)} \rightarrow (\tau_1 \times \tau_2) \; \text{list(n)}
\]
**Task 9** (2 points). Give a dependent type for \texttt{zip} using \(\Sigma\) types, and \(\Pi\) types if necessary (no code necessary, just the type).

**Solution:**

\[
\Pi n : \text{nat. } \Pi m : \text{nat. } \tau_1 \text{ list}(n) \to \tau_2 \text{ list}(m) \to \Sigma o : \text{nat. } (\tau_1 \times \tau_2) \text{ list}(o)
\]

**Solution:** Comments on tasks 8 and 9: in the end, I decided to give everyone 4 points for this problem, because clearly there was lots of confusion and I wasn’t sure about whether it was the greatest problem. Note, though, that the original type signatures presented were only meant as analogies to guide your thinking; the idea wasn’t to have you emulate them, or worse, try to emulate both of them for a single part!

The type signature with \texttt{unit} + \((\tau_1 \times \tau_2) \text{ list}\) was meant to correspond to task 8, not 9, since 8 is the one where it fails to typecheck if the lists aren’t exactly the same length. Also, since it’s supposed to fail statically, not dynamically (i.e. at typechecking, not at runtime), there’s no need for the result type to be \texttt{unit} + \((\tau_1 \times \tau_2) \text{ list}(n)\), which a bunch of people did even if they correctly identified \texttt{unit} as corresponding to task 8.

Also, you’re right if you think the dependent type in task 9 doesn’t seem very descriptive. Oh well.

### 4 Cut elimination (10 points)

In lecture, we proved

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

by nested induction first on the structure of \(A\), then on the derivation \(D\) of \(\Gamma \Rightarrow A\), then on the derivation \(E\) of \(\Gamma, A \Rightarrow C\).

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* \(\Sigma\). Sequents then have the form \(\Sigma; \Gamma \Rightarrow A\) where \(\Sigma\) is either empty or of the form \(a_1 : \tau_1, \ldots, a_n : \tau_n\), and \(\Gamma\) 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 \(\tau\) under signature \(\Sigma\). Our cut theorem is then

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

where in every case not involving quantifiers, the rules simply carry over the signature \(\Sigma\) 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 \( \Sigma; \Gamma \rightarrow A \), then for fresh \( c \), we have that
\[
\Sigma, c : \tau; \Gamma \rightarrow A
\]
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 \( A \).

**Solution:**

\[
\mathcal{E} = \begin{array}{c}
\Sigma, c : \tau; \Gamma, A \rightarrow A(c) \\
\Sigma; \Gamma, A \rightarrow \forall x : \tau. A(x) \end{array} \forall R
\]

\[
\mathcal{D} = \Sigma; \Gamma \rightarrow A
\]

Now
\[
\Sigma, c : \tau; \Gamma \rightarrow A \quad \text{by lemma 2 on } \mathcal{D}
\]
\[
\Sigma, c : \tau; \Gamma \rightarrow A(c) \quad \text{by IH on } A, \text{ above, } \mathcal{E}'
\]
where we note that \( \mathcal{D} \) has not grown bigger, since weakening yields a derivation with the same structure. Then
\[
\Sigma; \Gamma \rightarrow \forall x : \tau. A(x) \quad \text{by } \forall R
\]
which completes the proof of this case.

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

**Solution:**

\[
\mathcal{D} = \begin{array}{c}
\Sigma, c : \tau; \Gamma, \exists x : \tau. A(x), A(c) \rightarrow A \\
\Sigma; \Gamma, \exists x : \tau. A(x) \rightarrow A \end{array} \exists L
\]

\[
\mathcal{E} = \Sigma; \Gamma, \exists x : \tau. A(x), A \rightarrow C
\]

Now
\[
\Sigma, c : \tau; \Gamma, \exists x : \tau. A(x), A(c), A \rightarrow C \quad \text{by weakening and Lemma 2 on } \mathcal{E}
\]
\[
\Sigma, c : \tau; \Gamma, \exists x : \tau. A(x), A(c) \rightarrow C \quad \text{by IH on } A, \mathcal{D}', \text{ above}
\]
where we note that \( \mathcal{E} \) has not grown bigger because blah blah blah. Then
\[
\Sigma; \Gamma, \exists x : \tau. A(x) \rightarrow C \quad \text{by } \exists L
\]
which completes the proof of this case.
Solution: Comments on tasks 10 and 11: I think people generally got this stuff? (or I hope so). Minor quibbles: please use weakening! It’s important when you apply the IH/the cut theorem in general that the contexts/signatures match up exactly. Some people used weakening after using the IH to get their sequent into a form that they could apply $\forall R$ or $\exists L$ for. That is not what weakening is for! :(

Also, some people misapplied lemma 1. It’s very important to read things carefully and notice that lemma 1 “takes things out” of the signature.

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

$$B \land A \Rightarrow ((A \land B) \lor C) \land (D \lor (A \land B))$$

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)

Solution: I think almost everyone got these; the point is to prove $B \land A \Rightarrow A \land B$ just once and use cut to get it in. Eliminating cuts is great, but it can make things unpleasantly longer.