$\boxed{\text{CIS 500 Software Foundations}}$

Homework Assignment 4

λ-calculus, de Bruijn notation

Due: Monday, October 10, 2005, by noon

Submission instructions:

You must submit your solutions electronically (in ascii, postscript, or PDF format). Electronic solutions should be submitted following the same instructions as last time; these can be found at `http://www.seas.upenn.edu/~cis500/homework.html`. Do **not** email your solutions to us.

1 Exercise As quick self-checks, you should have done exercises 6.1.1, 6.2.2, 6.2.5, 6.2.7, and 6.3.1 in TAPL when you read Chapter 6. If you did not, do them now.[1] Do **not** turn in anything for this exercise.

2 Exercise We can extend the untyped λ-calculus (Figure 5-3 in TAPL) with a primitive notion of lists as follows. This is in contrast to encoding lists in the untyped λ-calculus, which you saw in exercise 5.2.8 from TAPL.

$$
\begin{array}{lcll}
\mathtt{t} & ::= & \ldots & \text{terms:} \\
& & [] & \text{empty list (nil)} \\
& & \mathtt{t} :: \mathtt{t} & \text{cons} \\
& & \mathtt{match}\ \mathtt{t}\ \mathtt{with}\ [] \Rightarrow \mathtt{t} \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t} & \text{case analysis on lists} \\
\\
\mathtt{v} & ::= & \ldots & \text{values:} \\
& & \mathtt{lv} & \text{list values} \\
\\
\mathtt{lv} & ::= & [] & \\
& & \mathtt{v} :: \mathtt{lv} &
\end{array}
$$

$$
\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1 :: \mathtt{t}_2 \rightarrow \mathtt{t}_1' :: \mathtt{t}_2}\ \text{E-Cons1}
\qquad
\frac{\mathtt{t}_2 \rightarrow \mathtt{t}_2'}{\mathtt{v}_1 :: \mathtt{t}_2 \rightarrow \mathtt{v}_1 :: \mathtt{t}_2'}\ \text{E-Cons2}
$$

$$
\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{(\mathtt{match}\ \mathtt{t}_1\ \mathtt{with}\ [] \Rightarrow \mathtt{t}_2 \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t}_3) \rightarrow (\mathtt{match}\ \mathtt{t}_1'\ \mathtt{with}\ [] \Rightarrow \mathtt{t}_2 \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t}_3)}\ \text{E-Match}
$$

$$
\frac{}{(\mathtt{match}\ []\ \mathtt{with}\ [] \Rightarrow \mathtt{t}_2 \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t}_3) \rightarrow \mathtt{t}_2}\ \text{E-MatchNil}
$$

$$
\frac{\text{side conditions: } \mathtt{x}_2 \notin FV(\mathtt{v}) \text{ and } \mathtt{x}_1 \neq \mathtt{x}_2}{(\mathtt{match}\ \mathtt{v} :: \mathtt{lv}\ \mathtt{with}\ [] \Rightarrow \mathtt{t}_2 \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t}_3) \rightarrow [\mathtt{x}_2 \mapsto \mathtt{lv}][\mathtt{x}_1 \mapsto \mathtt{v}]\mathtt{t}_3}\ \text{E-MatchCons}
$$

In the expression $\mathtt{match}\ \mathtt{t}\ \mathtt{with}\ [] \Rightarrow \mathtt{t}_1 \mid \mathtt{x}_1 :: \mathtt{x}_2 \Rightarrow \mathtt{t}_2$, we consider $\mathtt{x}_1$ and $\mathtt{x}_2$ to be bound in $\mathtt{t}_2$, just as we consider $\mathtt{x}$ to be bound in $\mathtt{t}$ when we write $\lambda\mathtt{x}.\mathtt{t}$. As in OCaml, $::$ associates to the right; for example, $\mathtt{t}_1 :: \mathtt{t}_2 :: \mathtt{t}_3$ is the same as $\mathtt{t}_1 :: (\mathtt{t}_2 :: \mathtt{t}_3)$.

1. In one step, what does $\mathtt{match}\ (\lambda\mathtt{x}.\mathtt{y}) :: (\lambda\mathtt{y}.\mathtt{x}) :: []\ \mathtt{with}\ [] \Rightarrow \mathtt{z} \mid \mathtt{a} :: \mathtt{b} \Rightarrow (\lambda\mathtt{x}.\mathtt{b}) :: (\lambda\mathtt{y}.\mathtt{a}) :: []$ evaluate to according to the above rules? Show the derivation.

[1] In general, you should be doing all the one $\star$ exercises when you read TAPL. They are designed to make sure you have a minimal understanding of what you're reading.

2. Read exercise 6.1.5 from TAPL. You may look at the solution in the back of the book. Now, extend the definitions for $\mathit{removenames}_\Gamma(\mathtt{t})$ and $\mathit{restorenames}_\Gamma(\mathtt{t})$ to include the primitive lists introduced above.

3. Now prove that $\mathit{restorenames}_\Gamma(\mathit{removenames}_\Gamma(\mathtt{t})) = \mathtt{t}$ for all $\mathtt{t}$, up to renaming of bound variables. Please turn in the entire proof. However, you may ignore primitive lists for this particular proof; that is, you may assume that $\mathtt{t}$ is a term from the following grammar:

$$\begin{array}{lcl} \mathtt{t} & ::= & \mathtt{x} \\ & & \mathtt{t}\ \mathtt{t} \\ & & \lambda\mathtt{x.t} \end{array}$$

4. Extend definition 6.2.4 in TAPL so that it is also defined on your nameless representation of lists. That is, give the definition of substitution on your nameless representation of lists.

5. Write down the nameless equivalents of all the evaluation rules above (i.e., E-Cons1, E-Cons2, etc.). For example, the version of E-AppAbs on p.81 of TAPL is the nameless equivalent of the E-AppAbs rule from Figure 5-3.

3 Exercise In this exercise, we're back to named λ-calculus terms. However, instead of the familiar call-by-value evaluation rules, we'll consider leftmost/outermost evaluation. This is called the "normal order" strategy in TAPL (see p.56). We can define this evaluation strategy by the following rules.

$$\frac{\mathtt{t_1} \to \mathtt{t_1}'}{\mathtt{t_1}\ \mathtt{t_2} \to \mathtt{t_1}'\ \mathtt{t_2}}\ \text{E-App1} \qquad \frac{\lambda\mathtt{x.t_1} \not\to}{(\lambda\mathtt{x.t_1})\ \mathtt{t_2} \to [\mathtt{x} \mapsto \mathtt{t_2}]\mathtt{t_1}}\ \text{E-App2} \qquad \frac{\mathtt{t_2} \to \mathtt{t_2}'}{\mathtt{x}\ \mathtt{t_2} \to \mathtt{x}\ \mathtt{t_2}'}\ \text{E-App3}$$

$$\frac{(\mathtt{s}\ \mathtt{t}) \not\to \qquad \mathtt{t_2} \to \mathtt{t_2}'}{(\mathtt{s}\ \mathtt{t})\ \mathtt{t_2} \to (\mathtt{s}\ \mathtt{t})\ \mathtt{t_2}'}\ \text{E-App4} \qquad \frac{\mathtt{t_1} \to \mathtt{t_1}'}{\lambda\mathtt{x.t_1} \to \lambda\mathtt{x.t}_1'}\ \text{E-Abs}$$

Here, we use the notation $\mathtt{t} \not\to$ to indicate that for all $\mathtt{t}'$, we cannot build a derivation of $\mathtt{t} \to \mathtt{t}'$.

1. Spend a few moments convincing yourself that these rules lead to reducing the leftmost, outermost redex in a term. Remember that "leftmost" and "outermost" describe the position of the redex when we view the term as an abstract syntax tree. Do **not** turn in anything for this problem.

2. We can modify the `eval1` function from Chapter 7 to implement these rules instead of the familiar call-by-value rules. Lucky for you, someone tried to do that already and produced the following code.

```
let rec eval1 ctx t = match t with
   (* First case. *)
   TmAbs(fi,x,t1) ->
      let t1' = eval1 ctx t1 in
      TmAbs(fi,x,t1')
 | TmApp(fi,t1,t2) ->
      (try
         (* Second case. *)
         let t1' = eval1 ctx t1 in
         TmApp(fi,t1',t2)
      with
         NoRuleApplies ->
            (* Third case. *)
            let t2' = eval1 ctx t2 in
            TmApp(fi,t1,t2'))
 | _ ->
      raise NoRuleApplies
```

Not so lucky for you, while this code compiles, it does contain some major mistakes in one of the cases (the comments indicate there are three cases in the code). Which case is that and how would you fix it? Do not modify an actual implementation for this problem. Just write out your fix to the code by hand; we will ignore *minor* syntax errors as long as your intent is clear (and correct).

4 Debriefing

1. How many hours did each person in your group spend on this assignment, including time taken to read TAPL?
2. Would you rate it as easy, moderate, or difficult?
3. Did everyone in your study group participate?
4. How deeply do you feel you understand the material it covers (0%–100%)?

If you have any other comments, we would like to hear them; please send them `cis500@cis.upenn.edu`.